

\$16.95

80

Robert T. Grauer

PRENTICE-HALL SERIES IN PERSONAL COMPUTING

TRS-80 COBOL

Robert T. Grauer, Ph.D.

University of Miami, Florida

PRENTICE-HALL, INC., ENGLEWOOD CLIFFS, NEW JERSEY 07632

Library of Congress Cataloging in Publication Data

Grauer, Robert T.
TRS-80 COBOL.

(Prentice-Hall series in personal computing)

Includes index.

1. TRS-80 (Computer)—Programming. 2. Basic
(Computer program language) I. Title. II. Series.
QA76.8.T18G7 1983 001.64'24 82-16638

ISBN 0-13-931212-9

ISBN 0-13-931204-8 (pbk.)

Editorial/Production Supervision
and Interior Design: *Lynn S. Frankel*
Cover Design: *Photo Plus Art*
Manufacturing Buyer: *Gordon Osbourne*

© 1983 by Tandy Corporation, Fort Worth, Texas 76102

All rights reserved. No part of this book
may be reproduced in any form or by any means
without permission in writing from the publisher.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-931212-9 {CASE}
ISBN 0-13-931204-8 {PAPER}

Prentice-Hall International, Inc., *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Editora Prentice-Hall do Brazil, Ltda., *Rio de Janeiro*
Prentice-Hall Canada, Inc., *Toronto*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Whitehall Books Limited, *Wellington, New Zealand*

*To my family
Marion, Benjy, and Jessica
with whom I share much more
than a home computer*

The following information is reprinted from COBOL Edition, 1965, published by the Conference on Data Systems Languages (CODASYL), and printed by the U.S. Government Printing Office:

Any organization interested in reproducing the COBOL report and specifications in whole or part, using ideas taken from this report as the basis for an instructional manual or for any other purpose, is free to do so. However, all such organizations are requested to reproduce this section as part of the introduction to the document. Those using a short passage, as in a book review, are requested to mention "COBOL" in acknowledgment of the source, but need not quote this entire section.

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the COBOL Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

Procedures have been established for the maintenance of COBOL. Inquiries concerning the procedures for proposing changes should be directed to the Executive Committee of the Conference on Data Systems Languages.

The authors and copyright holders of the copyrighted material used herein

FLOWMATIC (Trade mark of the Sperry Rand Corporation), Programming for the Univac (R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell.

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals of similar publications.

CONTENTS

PREFACE

xi

CHAPTER 1 INTRODUCTION

1

Overview 2
The First Problem 2
COBOL: A First Look 2
Elements of COBOL 4
Summary 8
True/False 8
Exercises 9

CHAPTER 2 FILE PROCESSING

11

Overview 12
Vocabulary 12
Employee Selection Problem 12
*Flowcharts • Pseudocode • Test Data • Implementation
in COBOL*
Summary 19
True/False 19
Exercises 20

CHAPTER	TRS-80 COBOL	23
3		
	Overview 24	
	COBOL Versus Machine Language 24	
	The TRS-80 Operating System 26	
	Compilation on the TRS-80 26	
	<i>Compiler Output</i>	
	Execution on the TRS-80 30	
	Obtaining Hard Copy 30	
	DIR and KILL Commands 30	
	The Source Program Editor — CEDIT 31	
	<i>A Sample Session</i>	
	Creating a COBOL Program 34	
	Creating a Data File 34	
	Summary 35	
	True/False 36	
	Exercises 36	
	Projects 38	
 CHAPTER	 THE COBOL LANGUAGE	 41
4		
	Overview 42	
	COBOL Notation 42	
	Identification Division 43	
	Environment Division 44	
	Data Division 44	
	<i>Picture Clause • Level Numbers • File Section •</i>	
	<i>Working-Storage Section • Value Clause</i>	
	Procedure Division 48	
	<i>ADD • SUBTRACT • MULTIPLY • DIVIDE • COMPUTE • READ</i>	
	<i>WRITE • OPEN • CLOSE • MOVE • Editing Numeric Data •</i>	
	<i>IF • PERFORM • STOP RUN</i>	
	Summary 59	
	True/False 64	
	Exercises 64	
	Projects 66	
 CHAPTER	 DEBUGGING	 69
5		
	Overview 70	
	Errors in Compilation 70	
	<i>A Second Example</i>	
	Errors in Execution 85	
	Summary 86	
	True/False 88	
	Exercises 89	

CHAPTER 6 ADVANCED FEATURES

97

Overview 98

IF 98

Compound Tests • Condition Name Tests (88-Level Entries) • Nested IFs

PERFORM 102

DISPLAY 104

ACCEPT 104

READ INTO 105

WRITE FROM 106

ROUNDED and SIZE ERROR Options 106

Duplicate Data Names 107

Qualification • CORRESPONDING

INSPECT 109

Tables 109

The OCCURS Clause • Processing a Table • REDEFINES Clause • Table Lookups

Summary 114

True/False 119

Exercises 120

Projects 122

CHAPTER 7 PROGRAMMING STYLE

125

Overview 126

Coding Standards 126

Choose Meaningful Names 127

Avoid Commas 128

Use Appropriate Comments 128

Eliminate 77-Level Entries 129

Space Attractively 129

Indent 130

Avoid Constants 131

Avoid Literals 132

Keep It Simple 133

Perform Paragraphs, not Sections 134

Restrict Subscripts to a Single Use 134

Use 88-Level Entries 135

Structured Programming 136

The Completed Program 138

Summary 143

True/False 143

Projects 143

CHAPTER	CONTROL BREAKS	145
8		
	Overview 146	
	Editing 146	
	<i>Signed Numbers</i>	
	Control Breaks 148	
	One-Level Control Breaks 149	
	Two-Level Control Breaks 154	
	Summary 159	
	True/False 160	
	Exercises 160	
	Projects 164	
CHAPTER	SUBPROGRAMS AND THE COPY STATEMENT	167
9		
	Overview 168	
	COPY Statement 168	
	Subprograms 169	
	A Complete Example 171	
	Summary 176	
	True/False 177	
	Exercises 178	
	Projects 182	
CHAPTER	TABLE PROCESSING	183
10		
	Overview 184	
	The OCCURS Clause 184	
	Table Lookups — A Review 186	
	Initializing Tables Dynamically 187	
	The SET Statement 188	
	A Complete Example 189	
	Two-Level Tables 196	
	<i>PERFORM/VARYING • A Complete Example</i>	
	Three-Level Tables 202	
	<i>PERFORM/VARYING</i>	
	Summary 204	
	True/False 205	
	Exercises 205	
	Projects 211	

CHAPTER 11	SEQUENTIAL FILE MAINTENANCE	213
	Overview 214	
	Concepts of File Maintenance 214	
	Murphy's Law 215	
	Requirements of the Maintenance Program 215	
	Pseudocode 217	
	Hierarchy Charts 218	
	Top Down Development 219	
	<i>A Stubs Program • The Completed Program</i>	
	Summary 229	
	True/False 230	
	Exercises 231	
 CHAPTER 12	 NONSEQUENTIAL FILE MAINTENANCE	 235
	Overview 236	
	Diskette Organization 236	
	Indexed Files 236	
	COBOL Requirements for Indexed Files 238	
	DECLARATIVES 240	
	Creating an Indexed File 242	
	Printing an Indexed File 242	
	Logical Requirements of the Maintenance Program 242	
	Updating an Indexed File 245	
	Summary 249	
	True/False 249	
	Exercises 250	
 APPENDIX A	 ANSWERS TO TRUE/FALSE AND EXERCISES	 255
 APPENDIX B	 RESERVED WORD LIST	 275
 APPENDIX C	 TRS-80 COBOL SYNTAX	 281
 APPENDIX D	 GLOSSARY	 297
	 INDEX	 303

PREFACE

COBOL is a higher level language, which in theory ought to make it machine independent. In practice, however, COBOL contains an Environment Division which makes it, to a limited extent, machine dependent. The problem is further compounded by various implementations of the ANS standard. Many vendors have included their own extensions, while others have failed to implement portions of the standard. In short, a competent COBOL programmer has to know *specifics of the implementation on his or her machine*.

Even if COBOL were truly compatible from one machine to the next, operating systems are totally different. It is one thing to write a sequential update in COBOL; it is quite another to create and access the data files which the program requires. Text editors also vary greatly, and the COBOL programmer must be knowledgeable in this area as well.

TRS-80 COBOL is written specifically for the Radio Shack Model II and III computers. The author believes that knowledge of COBOL, in and of itself, does not necessarily yield a complete programmer. Included, therefore, are specifics of the TRS-80 COBOL implementation, characteristics of the operating system (TRSDOS), and instruction on its text editor (CEDIT).

At the same time, *TRS-80 COBOL* is a *substantial* COBOL text covering all elements of the language, as implemented on the TRS-80 machines. The author adheres to sound programming practices, and uses *structured programming* exclusively. Pseudocode and hierarchy charts are emphasized, while the traditional flowchart is de-emphasized.

The author employs a "learn by doing" approach which stresses early access to the machine as well as constant exposure to complete COBOL programs. Every COBOL chapter contains at least one program to tie together the major points in that chapter.

Chapter 1 is a rapid introduction to COBOL, and the reader is exposed to a complete program almost immediately. The intent is not to master the myriad syntactical rules associated with the language, but rather to gain a conceptual understanding of what programming is all about. The chapter develops the fundamental concept that every computer program consists of three phases — input, processing, and output.

Chapter 2 develops concepts of file processing. A COBOL program is written to read an incoming file, select various records, and prepare a report. Associated topics include the use of flowcharts and/or pseudocode and the preparation of test data.

Chapter 3 deals exclusively with the TRS-80. It covers the COBOL text editor (CEDIT), elementary commands of the TRSDOS operating system, and use of the COBOL compile (RSCOBOL), and runtime (RUNCOBOL) modules. The chapter also discusses file name conventions, differentiating between source, object, and data files.

Chapter 4 returns to COBOL. It introduces the COBOL notation, then uses the material to explain a basic COBOL subset. The Data Division is covered in some depth, with emphasis on the File and Working-Storage sections, and group versus elementary items. The chapter ends with another complete program.

Chapter 5 debugs both compilation and execution errors. The program of Chapter 4 is rewritten to illustrate common compilation errors, and rewritten a second time to depict typical execution errors.

Chapter 6 introduces some advanced elements of the language. These include compound tests, condition names, nesting in an IF statement, as well as several formats of the PERFORM verb. The chapter presents basics of table processing, including the OCCURS and REDEFINES clauses, and rudiments of a sequential “table lookup”.

Chapter 7 discusses programming style, through a series of guidelines designed to make a COBOL program easy to follow. The chapter stresses that a well-written program must not only work, but in addition be easily read and maintained by someone other than the original author. The chapter formally defines “structured programming”, although every program from Chapter 1 on has been structured.

Chapter 8 introduces the concept of control breaks, one of the most frequent data processing applications. Two programs are developed for single and double control breaks, respectively. Each is accompanied with pseudocode and a hierarchy chart. The chapter also completes an earlier presentation of editing, including use of signed numbers and incorporates this material into the illustrative programs.

Chapter 9 covers subprograms and the COPY statement, both powerful COBOL techniques. Specifics of the TRS-80 implementation are shown in the programs at the chapter’s end.

Chapter 10 is a comprehensive treatment of table processing. The material on table lookups is reviewed from Chapter 6, then extended to cover variable-length tables, indexing, and the SET statement. Various techniques for table lookups and initialization are contrasted in a complete program. Two-level tables are covered in depth through a discussion of PERFORM/VARYING/AFTER, multiple OCCURS clauses, and a second program. The concepts are then extended to three-level tables.

Chapter 11 focuses on sequential file maintenance. It develops basic vocabulary; e.g., fixed versus variable-length records, blocked versus unblocked records, and so on. The emphasis, however, is on development of a nontrivial maintenance program, entirely through the structured method-

ology. Coverage includes top down development and testing, use of program stubs, pseudocode, and hierarchy charts.

Chapter 12 parallels Chapter 11, except for nonsequential maintenance. It presents concepts of indexed file organization and the associated COBOL elements. It then develops a nonsequential maintenance program with requirements similar to the one in Chapter 11.

Exercises are present at the end of each chapter. Of particular interest are the debugging problems at the ends of Chapters 8 through 12. Solutions to all exercises are included in Appendix A, but no peeking until you have made an honest attempt at achieving your own solution. In addition, most chapters have one or more programming projects, which *must* be attempted if the reader is to master the material.

ACKNOWLEDGMENTS

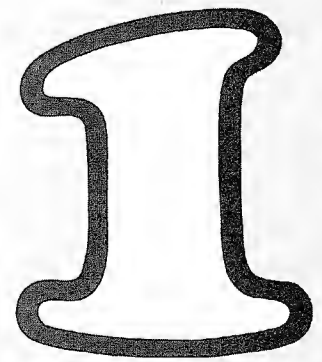
The author gratefully acknowledges permission from the Tandy Corporation to reprint copyrighted material. He is also appreciative of the help provided by several employees of the Tandy Corporation: Mr. Dave Carpenter, Mr. Bill Gattis, Ms. Barbara Kemp, Ms. Karen McGee, and Mr. Dennis Tanner.

At Prentice-Hall, the author wishes to thank his editor, Bernard Goodwin, for making this book possible. He expresses special thanks to Lynn Frankel, production editor, who brings a new meaning to the word "dedication." Herb Daehnke is to be commended for the artwork, as are proofreaders Mark Paris and Jackie Clark for their contribution to the quality of the book. Last, and certainly not least, thanks to Edith Butler and Sheila Grossman who patiently typed and retyped the manuscript.

SUPPORTING PRODUCTS OF THE TANDY CORPORATION

In addition to the Model II and/or Model III computer, and the associated COBOL compilers, the reader may wish to purchase a COBOL instructional diskette that contains a copy of every program in the book, and/or COBOL class notes for use at a Radio Shack Computer Center. (See product numbers 26-2702 and 26-2706 for the Model III. Check 26-2723 and 26-2724 for the Model II.)

ROBERT T. GRAUER



INTRODUCTION

OVERVIEW This book is about computer programming. In particular, it is about the TRS-80 computer and COBOL, a widely used programming language. Programming involves the translation of a precise means of problem solution into a form the computer can understand. Programming is necessary because, despite reports to the contrary, computers cannot think for themselves. Instead they do exactly what they have been instructed to do, and these instructions take the form of a computer program. The advantage of the computer stems from its speed and accuracy. It does not do anything that a human being could not do, if he or she were given sufficient time.

All computer applications consist of three phases: *input*, *processing*, and *output*. Information enters the computer, it is processed (i.e., calculations are performed) and the results are communicated to the user. Input can come from the TRS-80 keyboard, a diskette, a tape cassette, or any of a variety of other devices. Processing encompasses the logic to solve a problem, but in actuality all a computer does is add, subtract, multiply, divide, or compare. All logic stems from these basic operations, and the power of the computer comes from its ability to alter a sequence of operations based on the results of a comparison. Output can take several forms. It may consist of a computer printout, or it may be payroll checks, computer letters, mailing labels, etc.

We shall begin our study of computer programming by posing a simple problem for solution on the TRS-80. We move quickly into COBOL and examine a complete program. The reader may observe that this rapid entrance into COBOL is somewhat different from the approach followed by most books, but the author believes in learning by doing. There is nothing very mysterious about COBOL programming, so let's get started.

THE FIRST PROBLEM Let us pose a very simple problem; calculate the average of three test grades. In order to obtain the solution, the three grades must be added together and the sum divided by three.

If this problem is to be solved on a computer, one must provide for *the three phases of any computer application: input, processing, and output*. One has to enter the test grades into the computer, which in turn processes the data to obtain an average, and finally the computer has to display the calculated results.

COBOL: A FIRST LOOK Every COBOL program contains four divisions, which appear in specified order. These are:

**IDENTIFICATION
DIVISION**

This division contains the program and author's name. It can also contain other identifying information, such as date written, installation name, and so on.

**ENVIRONMENT
DIVISION**

This portion mentions the computer on which the program is to be compiled and executed (usually one and the same). It also specifies the input/output devices to be used by the program.

DATA DIVISION

This division describes the location of incoming and outgoing data. It can, for example, describe particular columns where output information is to appear in a report.

PROCEDURE DIVISION

This division contains the program logic, that is, the instructions the computer is to execute in solving the problem.

With this briefest of introductions, consider Figure 1.1, which is a complete program to obtain the average test score. The syntactical rules for

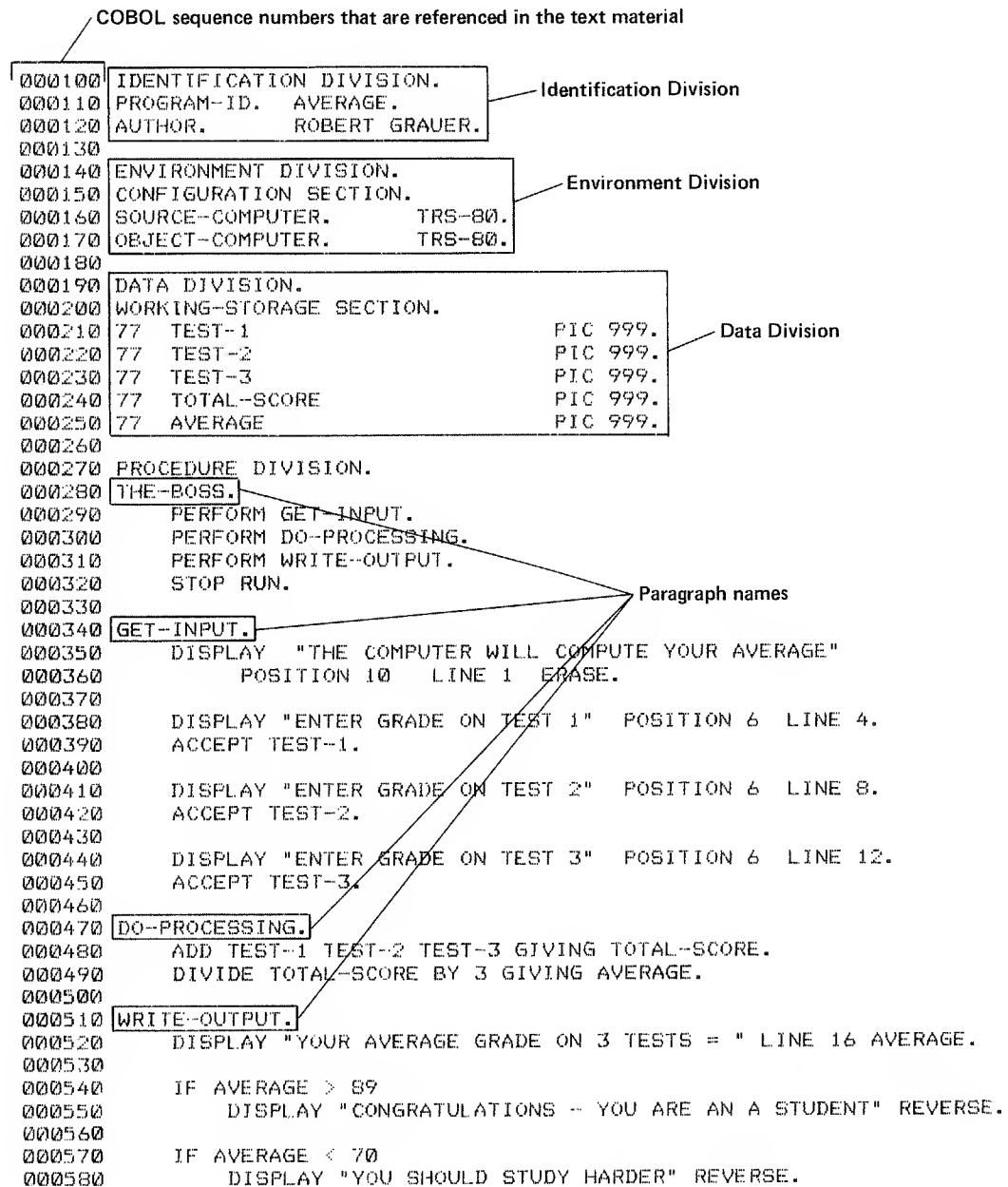


FIGURE 1.1 The first COBOL program

COBOL are very precise, and you are certainly not expected to remember them now. However, the author believes that immediate exposure to a computer program is extremely beneficial in terms of stripping away the mystical aura that too often surrounds programming.

Consider the Procedure Division of Figure 1.1, which contains the program's logic and consequently is the most important part of any COBOL program. The Procedure Division of Figure 1.1 begins on line 270. It is divided into four *paragraphs*, THE-BOSS, GET-INPUT, DO-PROCESSING, and WRITE-OUTPUT, beginning on lines 280, 340, 470, and 510, respectively.

The relationship of these paragraphs to each other is best explained by referring to the *hierarchy chart* of Figure 1.2. A hierarchy chart for a program is very much like an organization chart for a company. It shows which paragraph in a program is the "president" and which paragraphs are subordinates. As can be seen from Figure 1.2, GET-INPUT, DO-PROCESSING, and WRITE-OUTPUT are all subordinate to the paragraph, THE-BOSS.

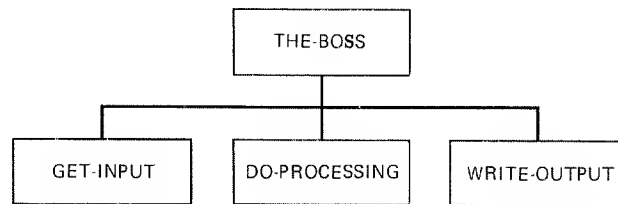


FIGURE 1.2 Hierarchy chart for the first COBOL program

Statements 290, 300, and 310 in Figure 1.1 are all PERFORM statements. The PERFORM verb in COBOL transfers control to the designated paragraph, which does its job, and on completion returns control to the statement following the original PERFORM. Look carefully at THE-BOSS paragraph, beginning in line 280. The first thing it does is invoke a subordinate, GET-INPUT, to obtain the test grades. Next, it calls a second subordinate, DO-PROCESSING, to compute the average, and finally a third subordinate to WRITE-OUTPUT. When the job of the last paragraph is completed, control returns to line 320, which stops the run and terminates the program.

Now that we have an appreciation for the *overall workings* of the program, let us consider the subordinate paragraphs. To carry the company organization analogy a bit further, one might say we are looking at the "job descriptions" of GET-INPUT, DO-PROCESSING, and WRITE-OUTPUT.

The GET-INPUT paragraph consists entirely of DISPLAY and ACCEPT statements. The former prints a message on the terminal, e.g., "ENTER GRADE ON TEST 1" (in line 380); the latter waits for the user to comply and stores the result in the computer's memory.

The DO-PROCESSING paragraph adds the input obtained by its colleague, GET-INPUT, and stores the result as TOTAL-SCORE (line 480). It then divides TOTAL-SCORE by 3 and obtains AVERAGE in line 490.

WRITE-OUTPUT displays the average in line 520. This paragraph also employs two IF statements in lines 540 and 570, which display additional messages, depending on the calculated results.

ELEMENTS OF COBOL

Although the reader is certainly *not* yet expected to be able to write a COBOL program, he or she may be able to intuitively follow simple programs like Figure 1.1. This section begins a formal discussion of COBOL so that one will eventually be able to write an entire program.

COBOL is comprised of six language elements: reserved words, programmer-supplied names, literals, symbols, level numbers, and pictures. Every COBOL statement contains at least one *reserved word*, which gives the entire statement its meaning. Reserved words have special significance and are used in a rigidly prescribed manner. They must be spelled correctly or else COBOL will not recognize them. The list of reserved words varies from computer to computer, and the TRS-80 list is given in Appendix B. The beginner is urged to refer frequently to this appendix for two reasons: (1) to ensure the proper spelling of reserved words used in his or her program, and (2) to avoid the inadvertent use of reserved words as *programmer-supplied names*.

The programmer supplies his own names for files, paragraphs, and data names. A paragraph name is a tag to which the program refers, for example, GET-INPUT or DO-PROCESSING in Figure 1.1. Data names are the elements on which instructions operate, for example, TEST-1 and AVERAGE in Figure 1.1. A programmer chooses his own names within the following rules:

1. A programmer-supplied name can contain the letters A to Z, the digits 0 to 9, and the hyphen (-). No other characters are permitted, not even blanks.
2. Data names must contain at least one letter. Paragraph names may be all numeric.
3. A programmer-supplied name cannot begin or end with a hyphen.
4. Reserved words may not be used as programmer-supplied names.
5. Programmer-supplied names must be 30 characters or less.

The following examples should clarify the rules associated with programmer-supplied names:

<i>Programmer-Supplied Name</i>	<i>Explanation</i>
SUM-OF-X	Valid.
SUM OF X	Invalid: contains blanks.
SUM-OF-X-	Invalid: ends with a hyphen.
SUM-OF-ALL-THE-XS	Valid.
SUM-OF-ALL-THE-XS-IN-ENTIRE-PROGRAM	Invalid: more than 30 characters.
GROSS-PAY-IN-\$	Invalid: contains character other than letter, number, or hyphen.
12345	Valid as paragraph name, but invalid as a data name.

A *literal* is an exact value or constant. It may be numeric, that is, a number, or non-numeric, that is, enclosed in quotes. Literals appear throughout a program, for example in lines 570 and 580 of Figure 1.1:

```
IF AVERAGE < 70
  DISPLAY "YOU SHOULD STUDY HARDER".
```

In line 570 the *numeric* literal 70 is compared to the programmer-defined name AVERAGE. In the next statement, the *non-numeric* literal, YOU SHOULD STUDY HARDER, is displayed on the terminal.

Non-numeric literals are contained in quotes and may be up to 120 characters in length. Anything, including blanks, numbers, or reserved words,

may appear in the quotes and be part of the literal. *Numeric literals* can be up to 18 digits and may begin with a leading (left-most) plus or minus sign. The latter may contain a decimal point, but cannot end on a decimal point. Examples are shown:

<i>Literal</i>	<i>Explanation</i>
123.4	Valid numeric literal.
"123.4"	Valid non-numeric literal.
"IDENTIFICATION DIVISION"	Valid non-numeric literal.
123.	Invalid numeric literal: cannot end on a decimal point.
123-	Invalid numeric literal: the minus sign must be in the left-most position.

Symbols are of three types, punctuation, arithmetic, and conditional, and are contained in Table 1.1.

TABLE 1.1
COBOL SYMBOLS

<i>Category</i>	<i>Symbol</i>	
Punctuation	.	Denotes end of COBOL entry
	,	Delineates clauses
	;	Delineates clauses
	" "	Sets off non-numeric literals
	()	Encloses subscripts or expressions
Arithmetic	+	Addition
	-	Subtraction
	*	Multiplication
	/	Division
Conditional	=	Equal to
	<	Less than
	>	Greater than

The use of conditional and arithmetic symbols is described in detail later in the text, beginning in Chapter 4. Commas and semicolons are used to improve the readability of a program, and their omission (or inclusion) does not constitute an error. Periods, on the other hand, should be used after a sentence, and their omission could cause difficulty.

The period also has special significance with respect to certain COBOL verbs, for example, the IF statement. Simply stated, the period terminates the effect of the verb. Consider the difference between:

Example 1

```
IF TITLE = "PROGRAMMER"
  ADD 1 TO NUMBER-OF-PROGRAMMERS
  MOVE CARD-NAME TO OUTPUT-AREA.
```

no period

Example 2

```
IF TITLE = "PROGRAMMER"
  ADD 1 TO NUMBER-OF-PROGRAMMERS.
  MOVE CARD-NAME TO OUTPUT-AREA.
```

extra period

The two examples contain almost identical COBOL except for the extra period in example 2. (They are, however, indented differently to highlight the difference in action caused by the extra period.) When an IF condition is satisfied, all action between the IF and the period is taken. Hence, in example 1, both the ADD and MOVE will be executed if TITLE = "PROGRAMMER". However, in example 2, *only* the ADD will be executed if TITLE = "PROGRAMMER", and the MOVE will be executed *regardless* of the TITLE. Much more is said about the IF in Chapter 4.

Level numbers are used in defining data names. We shall learn in Chapter 2 that level numbers may go from 01 to 49, and can also include the special number 77. (Only the latter appears in Figure 1.1.)

Pictures are used in the Data Division to describe the nature of incoming or outgoing data. A picture of 9's means the entry is numeric, a picture of A's implies the entry is alphabetic, and a picture of X's means the entry is alphanumeric and can contain letters, numbers, and special characters. Note, however, that alphabetic pictures are seldom used; that is, even names can contain apostrophes or hyphens, which are alphanumeric rather than alphabetic in nature. Lines 210 through 250 of Figure 1.1 illustrate the use of the PICTURE clause. (The reserved word PICTURE can be abbreviated as PIC.)

Level numbers and pictures are discussed more fully in Chapter 4, under the Data Division. The reader may review Figure 1.1 and identify the various COBOL elements. As a final aid, consider Figure 1.3, which offers further intuitive explanation of the program.

000100	Header for IDENTIFICATION DIVISION.
000110	Names the program as AVERAGE.
000120	Identifies the author as Robert Grauer.
000140	Header for ENVIRONMENT DIVISION.
000150	Beginning of CONFIGURATION SECTION.
000160	Identifies TRS-80 as SOURCE-COMPUTER, the machine on which the program will compile. (Chapter 3 describes the compilation concept in detail.)
000170	Identifies TRS-80 as OBJECT-COMPUTER, the machine on which the program will execute.
000190	Header for DATA DIVISION.
000200	Identifies the WORKING-STORAGE SECTION.
000210-000250	Defines programmer-supplied data names which will be used in the program. Note that all five data names are 77-level entries, and further that each is a three position numeric field.
000270	Header for PROCEDURE DIVISION.
000280	Signals the first, and controlling, paragraph in the PROCEDURE DIVISION.

FIGURE 1.3 Line by line explanation of Figure 1.1

000290-00310	PERFORM statements which transfer control to lower level paragraphs for input, processing, and output, respectively.
000320	STOP RUN statement terminates program execution.
000340	GET-INPUT signals the beginning of the paragraph which obtains input. (Notice how lines 350 through 450 are indented under the paragraph name of line 340.)
000350-000450	DISPLAY and ACCEPT statements which prompt the user for input and receive same. The DISPLAY statements also contain specific line and position references where the displayed material is to appear.
000470-000490	The DO-PROCESSING paragraph which calculates the average of three test grades.
000510-000580	The WRITE-OUTPUT paragraph which displays the calculated result on the terminal.

FIGURE 1.3 *Continued*

SUMMARY This chapter presented a rapid introduction into COBOL. The reader was shown a completed program and given only a brief intuitive explanation. Nevertheless, the reader may have gained a basic understanding of computer programming and the fundamental concept that all applications consist of input, processing, and output.

Every COBOL program contains four divisions: Identification, Environment, Data, and Procedure, in that order. The latter contains a program's logic and is most important. A hierarchy chart, analogous to a company's organization chart, is useful in explaining the relationship among paragraphs in the Procedure Division.

The COBOL language is comprised of six elements. These are: reserved words, programmer-supplied names, literals, symbols, level numbers, and pictures. The chapter contained a formal introduction to COBOL, covering syntactical rules for these elements.

TRUE/FALSE

1. A COBOL program can run on a variety of computers.
2. The divisions of a COBOL program may appear in any order.
3. Non-numeric literals may not contain numbers.
4. Numeric literals may contain letters.
5. The picture clause indicates the type of data; e.g., numeric or alphanumeric.
6. The ACCEPT statement waits for a user response.
7. The DISPLAY statement prints a message on the CRT.
8. A data name cannot contain any characters other than a letter or a number.
9. The period has no effect on an IF statement.
10. Reserved words may be used as data names.

EXERCISES

1. With respect to the COBOL program of Figure 1.1,
 - (a) Identify five reserved words.
 - (b) Identify five programmer-supplied names.
 - (c) Identify three non-numeric literals.
 - (d) Identify three numeric literals.
 - (e) Identify two conditional symbols.
 - (f) Identify one level number.
 - (g) Identify one picture clause.
 - (h) Identify five Procedure Division verbs.
2. Modify Figure 1.1 so that:
 - (a) The message, "YOU SHOULD STUDY HARDER", will appear only if the average is less than 60.
 - (b) The message, "CONGRATULATIONS — YOU ARE AN A STUDENT", will never appear.
 - (c) The average is based on four grades instead of three.
3. Classify the following entries as being valid or invalid literals. For each valid entry, indicate if it is numeric or non-numeric; for each invalid entry, indicate the reason why it is invalid.
 - (a) 567
 - (b) 567.
 - (c) -567
 - (d) +567
 - (e) FIVE-SIX-SEVEN
 - (f) "567."
 - (g) "FIVE SIX SEVEN"
 - (h) "-567"
 - (i) 567-
 - (j) 567+
 - (k) "567+"
4. Indicate whether the following entries are valid as data names. If any entry is invalid, state the reasons.
 - (a) NUMBER-OF-TIMES
 - (b) DATE
 - (c) 12345
 - (d) ONE TWO THREE
 - (e) IDENTIFICATION-DIVISION
 - (f) IDENTIFICATION
 - (g) HOURS-
 - (h) GROSS-PAY-IN-DOLLARS
 - (i) GROSS-PAY-IN-\$

23

**FILE
PROCESSING**

OVERVIEW The COBOL program of Chapter 1 dealt with *limited* I/O (Input/Output). The person executing the program had to input values for three test grades through the keyboard, while the output was displayed on the CRT. This approach, using low speed I/O devices, is suitable in situations with small amounts of data, but unacceptable in commercial environments with large volumes of data.

This chapter discusses how to process substantial quantities of data. It begins with definition of basic terms: *field*, *record*, and *file*. It develops the logic necessary for file processing and presents both *flowcharts* and *pseudo-code*. Finally, it contains a complete COBOL program encompassing this logic, and discusses the COBOL elements required for file processing. These include the SELECT, FD, OPEN, CLOSE, READ, and WRITE statements.

VOCABULARY A *record* is a set of facts about a logical entity. An employee record, for example, might contain the employee's name, title, salary, and date of birth. A student record could contain the student's name, address, grade point average, year in school, and major. Each fact, e.g., name, title, salary, and date of birth is known as a *field*. Each employee record, therefore, has four fields, and each student record has five (name, address, grade point average, year in school, and major).

A *file* is a set of records. In a company of 1,000 employees, there would be 1,000 employee records (each with four fields) but *only one* employee file which contains all 1,000 records. With this as background, consider a simple example of file processing.

EMPLOYEE SELECTION PROBLEM Let us assume that we are to process a file of employee records and print the name and salary of any individual who is both a programmer and less than 30 years old. Let us further assume that the employee file contains sufficient data to solve the problem (i.e., each employee record has four fields: name, age, job title, and salary). Our program is to examine the first record and determine if the employee is qualified, then check the second record, and so on until all records have been processed. In other words, we are to develop a program that adheres to the following logic:

1. Read an employee record and stop when all records have been read.
2. If the employee whose record was just read is not a programmer, go back to step 1.
3. If the employee whose record was just read is 30 or over, go back to step 1.
4. Print relevant information for any employee reaching this point.
5. Go back to step 1.

Flowcharts

The first step in writing a program is to develop the logic the program will follow. The result of this effort can be a *flowchart*, which is nothing more than a pictorial representation of the logic inherent in a computer program.

A flowchart to determine the programmers under 30 is shown in Figure 2.1. The flowchart contains blocks with different shapes, where the shape implies the nature of the process. In particular, ellipses indicate the logi-

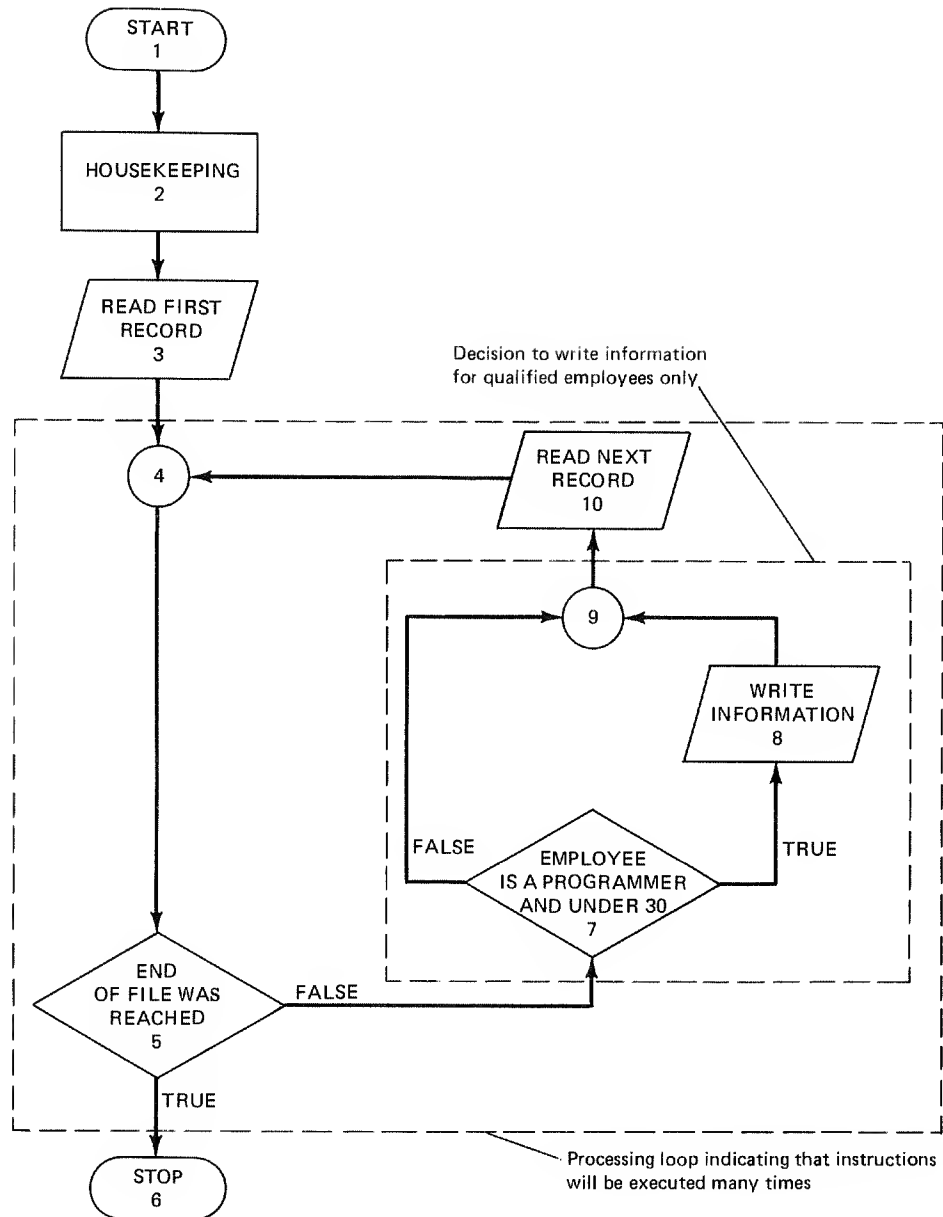


FIGURE 2.1 Flowchart to select programmers under 30

cal beginning or end of the entire flowchart, diamond-shaped boxes indicate a decision, parallelograms denote input/output operations, and rectangles denote straightforward processing. (All shapes conform to American National Standards Institute (ANSI) conventions and are used universally.)

The logic in Figure 2.1 should prove easily understandable, but Table 2.1 may provide additional insight. There is, however, one complication, the presence of *two* read blocks (blocks 3 and 10) and the “end of file” test in block 5. The necessity for these blocks is mandated by the nature of the read instruction. The function of a read is to obtain a record, but there will always be a point when a read is attempted and no record is found; that is, all the records have already been read. Since one does not know in advance how many records a file contains, *the read instruction must also signal the “end-of-file” condition*. Thus, if a file contains *two* records, it is actually

TABLE 2.1
BLOCK BY BLOCK EXPLANATION OF FLOWCHART IN FIGURE 2.1

<i>Block Number</i>	<i>Type</i>	<i>Explanation</i>
1	Terminal	Every flowchart contains a START block to indicate where program flow begins
2	Processing	As a rule, most programs require some initial processing known as housekeeping. Writing a page heading at the start of a report is a good example.
3	I/O	Reads the first record only. Note that, if the file is empty, execution of block 5 will cause the program to fall through without entering blocks 7 to 10 inclusive. The presence of this <i>initial read</i> is often bothersome to students when they first confront it. Its presence, however, is mandated by the nature of the read instruction and is further explained in the section on sample data.
4	Connector	Serves as an entry point into the loop which processes employee records.
5	Decision	Tests whether the end-of-file condition has been reached. If so, control goes to block 6, if not, control goes to block 7.
6	Terminal	Signals the end of processing.
7	Decision	Tests whether the employee currently being processed is a programmer and under 30. If so, control passes to block 8. If not, control passes to block 9.
8	I/O	Writes information on any employee reaching this point.
9	Connector	Terminates the decision making process.
10	I/O	Reads the next record, after which the connector in block 4 is reentered, and the end-of-file condition is tested in block 5. Note that when the end of file is sensed in block 10, control passes to blocks 4, 5, and 6, consecutively.

read *three* times (once for each record, and once to indicate that all records have been read). In similar fashion, a file containing N records is read $N+1$ times.

With this in mind, consider Figure 2.1 and assume there are only two records in the file, A and B. Record A is read by block 3. The end of file has not been reached, so blocks 7 through 9 are executed for record A. Record B is read in block 10. Again, the end of file has not been reached, so blocks 7 through 9 are executed for record B. When block 10 is executed a second time, the end of file is reached. Hence, the next execution of block 5 falls through to STOP. In summary, three reads have been executed; two records were processed, and the third read registered the end of file.

Pseudocode

The purpose of a flowchart is to convey the logic of a computer program in an easily understood form. An alternative technique, which has gained popularity in recent years, is *pseudocode* (also known as *structured English*). This method uses English statements in the form of instructions similar to those of a computer language, to describe logic. Pseudocode, however, is not bound by the formal syntactical rules followed by all programming languages. Nor is it bound by rules for indentation, which is done strictly at the discretion of the person using it. An example of pseudocode to describe the programmer problem is shown in Figure 2.2. As can be seen, pseudocode is

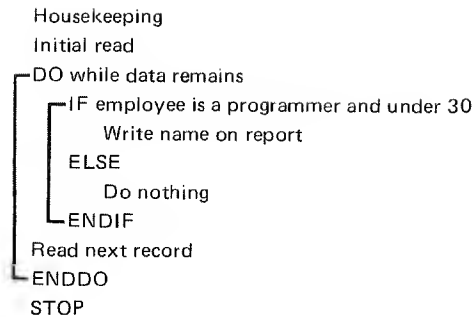


FIGURE 2.2 Pseudocode to select programmers under 30

considerably more succinct than a flowchart and just as easy to follow. Consequently, it is the distinct preference of this author.

Test Data

Figure 2.3 contains sample data for the flowchart of Figure 2.1. Consider what happens when these data are run through the flowchart.

JOHN DOE	ANALYST	35	23000	Rejected due to age
PEGGY WILCOX	PROGRAMMER	31	19000	
JOHN SMITH	PROGRAMMER	24	15000	
SHEILA LEVINE	PROGRAMMER	29	19000	
MARSHAL CRAWFORD	MANAGER	33	28000	
STANLEY STEAMER	PROGRAMMER	22	39000	Selected record
BENJAMIN LEE	PROGRAMMER	26	12000	
DICK PERSNICKETY	PROGRAMER	28	19000	
MARION MILGROM	JR. PROG	24	10000	Rejected because of title

FIGURE 2.3 Test data

The START and HOUSEKEEPING blocks are entered with no great effect. The first record, John Doe, is read by block 3. The end of file is not reached, so control passes to block 7, the test for a programmer under 30. Since John Doe is an analyst, rather than a programmer, control flows to the connector in block 9. The second record, Peggy Wilcox, is read in block 10. The end of file test is not met in block 5, so control passes again to the test in block 7. Ms. Wilcox is a programmer but is over 30, so control goes again to the connector in block 9, and finally to the read in block 10. John Smith is read in, and passes the programmer and age tests to reach the WRITE block (8). At this point, John Smith's name is written to a report, and the process continues.

Assume that the flowchart of Figure 2.1 was translated into a computer program, that the program was executed for the data shown in Figure 2.3, and that the report of Figure 2.4 resulted. The reader should carefully examine the input data and determine why various records did or did not appear in the resulting report.

Note in particular the absence of Dick Persnickety from Figure 2.4. At first glance it appears that an error has been made. This employee is a programmer and only 28 years old. Why then was the record omitted? Look carefully at how programmer is spelled; one "m" as opposed to two "m"s for the selected records. A human being knows that either spelling is acceptable, but the computer follows instructions to the letter. Apparently, it was

Spelled exactly as it appears in program of Figure 2.5

SALRY REPORT FOR PROGRAMMERS UNDER 30		
JOHN SMITH	24	15000
SHEILA LEVINE	29	19000
STANLEY STEAMER	22	39000
BENJAMIN LEE	26	12000

FIGURE 2.4 Printed report

not programmed to recognize the alternate spelling, and hence Persnickety was rejected.

The word “salary” is misspelled in the report of Figure 2.4. This is also indicative of the fact that *computers execute programs without regard to their correctness*. If a programmer misspells a word, and subsequently directs the computer to write that word, it will be written incorrectly. Mistakes of this kind, however, tend to be less severe than logic errors, and programmers are justifiably more concerned with logic rather than spelling. Users, on the other hand, will be concerned with spelling, choice of verbiage, formatting, and the like.

Implementation in COBOL

Now that we have developed the logic necessary for file processing, we can implement the flowchart of Figure 2.1 in COBOL. The resulting program is shown in Figure 2.5. The reader may already understand most of this program from the material in Chapter 1. Nevertheless, we discuss those statements in Figure 2.5 which pertain to file processing, and consequently did not appear in the first chapter. The discussion will again aim at conceptual understanding rather than detailed memorization. (Fear not, however; in Chapter 4 we provide a formal discussion of all COBOL elements covered to date.)

Line 190 of Figure 2.5 introduces the INPUT-OUTPUT SECTION of the Environment Division, which is required for file processing. Every file used in the program is first defined in a SELECT statement. Note well the presence of two SELECT statements, indicating two files. The first references EMPLOYEE-FILE, which contains the test data of Figure 2.3. The second, perhaps unexpected, file is PRINT-FILE and contains the report of Figure 2.4. *The SELECT statement ties programmer-chosen file names, e.g., EMPLOYEE-FILE and PRINT-FILE, to TRS-80 system names, e.g., FIRSTTRY/DAT and FIRSTTRY/TXT, which exist on a diskette.* (More on this in Chapter 3.)

Any file mentioned in a SELECT statement is further defined in an FD (file description) appearing in the FILE SECTION of the Data Division. The FD for EMPLOYEE-FILE extends from lines 280 through 310. It indicates standard labels, meaning the file information is in the usual TRS-80 format. It states that the record length is 80 characters (note well that the picture clauses sum to 80) and finally that the record name is EMPLOYEE-RECORD. Observe also that the record length for PRINT-FILE is 132 characters, the normal length for a printer.

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM ID.  FIRSTTRY.
000120 AUTHOR.      MARION MILGROM.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER.  TRS-80.
000170 OBJECT-COMPUTER. TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210     SELECT EMPLOYEE-FILE
000220     ASSIGN TO INPUT "FIRSTTRY/DAT".
000230     SELECT PRINT-FILE
000240     ASSIGN TO PRINT "FIRSTTRY/TXT".
000250
000260 DATA DIVISION.
000270 FILE SECTION.
000280 FD EMPLOYEE-FILE
000290     LABEL RECORDS ARE STANDARD
000300     RECORD CONTAINS 80 CHARACTERS
000310     DATA RECORD IS EMPLOYEE-RECORD.
000320 01 EMPLOYEE-RECORD.
000330     05 EMP-NAME PIC X(20).
000340     05 EMP-TITLE PIC X(10).
000350     05 EMP-AGE PIC 99.
000360     05 FILLER PIC XX.
000370     05 EMP-SALARY PIC 9(5).
000380     05 FILLER PIC X(41).
000390
000400 FD PRINT-FILE
000410     LABEL RECORDS ARE STANDARD
000420     RECORD CONTAINS 132 CHARACTERS
000430     DATA RECORD IS PRINT-LINE.
000440 01 PRINT-LINE.
000450     05 PRINT-NAME PIC X(25).
000460     05 FILLER PIC XX.
000470     05 PRINT-AGE PIC 99.
000480     05 FILLER PIC X(3).
000490     05 PRINT-SALARY PIC 9(5).
000500     05 FILLER PIC X(95).
000510
000520 WORKING-STORAGE SECTION.
000530 77 WS-DATA-REMAINS-SWITCH PIC X(3) VALUE SPACES.
000540
000550 PROCEDURE DIVISION.
000560 SELECT-PROGRAMMERS.
000570     OPEN INPUT EMPLOYEE-FILE
000580     OUTPUT PRINT-FILE.
000590     MOVE SPACES TO PRINT-LINE.
000600     MOVE "SALRY-REPORT FOR PROGRAMMERS UNDER 30" TO PRINT-LINE.
000610     WRITE PRINT-LINE
000620     AFTER ADVANCING 2 LINES.
000630     READ EMPLOYEE-FILE
000640     AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
000650     PERFORM PROCESS-RECORDS
000660     UNTIL WS-DATA-REMAINS-SWITCH = "NO".
000670     CLOSE EMPLOYEE-FILE
000680     PRINT-FILE.
000690     STOP RUN.
000700
000710 PROCESS-RECORDS.
000720     IF EMP-TITLE = "PROGRAMMER" AND EMP-AGE < 30
000730     MOVE SPACES TO PRINT-LINE
000740     MOVE EMP-NAME TO PRINT-NAME
000750     MOVE EMP-AGE TO PRINT-AGE
000760     MOVE EMP-SALARY TO PRINT-SALARY
000770     WRITE PRINT-LINE
000780     AFTER ADVANCING 2 LINES.
000790
000800     READ EMPLOYEE-FILE
000810     AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.

```

Required for file processing

Filename also specified in FD, OPEN, and CLOSE statements

FD is required for every file

EMP-NAME is in positions 1-20

EMP-AGE is in positions 31 and 32

PICTURE clauses sum to 132

Files are opened before processing

Salary is misspelled

Initial read

Files are closed prior to termination

Executed only when IF is satisfied

Single period terminates IF statement

Reads every record but the first

FIGURE 2.5 File processing program

IDENTIFICATION DIVISION.

PROGRAM-ID. B character name.
AUTHOR. your name.

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER. TRS-80.

OBJECT-COMPUTER. TRS-80.

INPUT-OUTPUT SECTION.

FILE-CONTROL.

SELECT statements are related
to TRS-80 data files and are
further explained in Chapter 3.

SELECT STUDENT-FILE ASSIGN TO
SELECT PRINT-FILE ASSIGN TO

DATA DIVISION.

FILE SECTION.

FD STUDENT-FILE

 LABEL RECORDS ARE OMITTED

 RECORD CONTAINS 80 CHARACTERS

 DATA RECORD IS STUDENT-RECORD.

01 STUDENT-RECORD.

 05 etc.

FD PRINT-FILE

 LABEL RECORDS ARE OMITTED

 RECORD CONTAINS 132 CHARACTERS

 DATA RECORD IS PRINT-LINE.

01 PRINT-LINE.

 05 etc.

WORKING-STORAGE SECTION.

77 EOF-SWITCH PIC XXX VALUE SPACES.

PROCEDURE DIVISION.

MAINLINE.

 OPEN INPUT STUDENT-FILE OUTPUT PRINT-FILE.

 READ STUDENT-FILE
 AT END MOVE "NO" TO EOF-SWITCH.

Initial (priming) READ
which is executed once.

 PERFORM PROCESS-RECORDS
 UNTIL EOF-SWITCH = "NO".

 CLOSE STUDENT-FILE, PRINT-FILE.
 STOP RUN.

PROCESS-RECORDS.

 your logic here

Last statement of performed
routine is a second READ.

 READ STUDENT-FILE
 AT END MOVE "NO" TO EOF-SWITCH.

FIGURE 2.6 Skeletal COBOL outline for file processing

The Procedure Division *opens* each file (lines 570 and 580) prior to actual processing. Lines 590 through 620 collectively write the heading line of Figure 2.4. The WRITE statement itself references PRINT-LINE, which is the *record* name defined in the FD for PRINT-FILE in line 430.

Lines 630 and 640 correspond to the “initial read” of block 3 in the flowchart of Figure 2.1. The READ of line 630 references EMPLOYEE-FILE, which appeared previously in SELECT, FD, and OPEN statements.

The PERFORM statement includes an UNTIL clause, which executes the paragraph PROCESS-RECORDS an indeterminate number of times *until* the value of WS-DATA-REMAINS-SWITCH = “NO”. The last statement of the paragraph PROCESS-RECORDS is another READ (block 10 in the flowchart) which accesses the next employee record in the file. If, however, the “end of file” is reached, then the AT END clause sets the value of WS-DATA-REMAINS-SWITCH to “NO”, which terminates the PERFORM and returns control to the statement under the PERFORM. Both files are *closed* in lines 670–680, and processing terminates.

SUMMARY This chapter made the critical transition from programs containing limited input/output to business applications with large volumes of data. It began by defining the terms field, record, and file, and developed the logic necessary for elementary file processing. Flowcharts and pseudocode were presented as alternative means of expressing logic.

The essence of the material is in the relationship of Figures 2.1, 2.3, and 2.4. The first contained a flowchart for the “Employee Selection Problem”, the second contained test data, and the third the resulting report. It is critical that the reader be able to *relate the output of Figure 2.4 to the input of Figure 2.3*.

Figure 2.5 contained the COBOL program to implement the logic of the flowchart. Attention was focused on the file processing aspects of COBOL; specifically, the SELECT, FD, OPEN, CLOSE, READ, and WRITE statements of Figure 2.5. The reader should also be able to tie the program of Figure 2.5 to the flowchart which preceded it. Finally, Figure 2.6 contains a skeletal outline of a COBOL program for file processing, which should prove useful as you begin to write your own programs.

TRUE/FALSE

1. A file is a set of records.
2. A record is a set of files.
3. The computer is a perfect speller and automatically corrects spelling errors.
4. A field contains one or more records.
5. Pseudocode must be written according to precise syntactical rules.
6. Pseudocode serves the same function as a flowchart.
7. A COBOL program often contains two distinct read statements.
8. The read statement typically contains an AT END clause.
9. A file name may not appear in more than two statements in the same program.
10. Every program must have a file section.

EXERCISES

1. Given the following data as input:

<i>Name</i>	<i>Location</i>	<i>Years of Service</i>	<i>Salary</i>
J. Anderson	Boston	4	\$40,000
V. Barbarino	New York	12	24,000
A. Horshack	N.Y., N.Y.	8	26,000
G. Kotter	Los Angeles	7	29,000
F. Unger	Chicago	9	18,000
O. Madison	Boston	7	34,000
V. Albright	N.Y.	11	26,000
M. Welby	New York	3	42,000

- What problems, if any, do you see in constructing a flowchart, and an eventual program, to determine the employees from New York with at least four years of service?
 - Develop a flowchart and corresponding pseudocode to select employees with the qualifications from part (a). (Use Figures 2.1 and 2.2 as a guide.)
 - Run the data through your flowchart. Which employees qualify? Which fail the service test? Which fail the location test?
 - Could you modify your flowchart to include only employees 30 years or older? Why or why not?
2. National Widgets is seeking a plant manager in Columbus, Ohio. It is looking to promote an individual from within the corporation, rather than hire from outside. The selected employee must have previous manufacturing experience and at least five years service with the company. Your programming manager has drawn a flowchart of this problem that will print the name of any qualified individual, as well as the number of qualified individuals. Unfortunately, he left it on his dining room table at home, and his two-year old daughter, Jessica, got to it first with a pair of scissors. Fortunately, he was able to gather the pieces (Figure 2.7) before Jessica could do further damage. Rearrange the pieces into a correct flowchart.

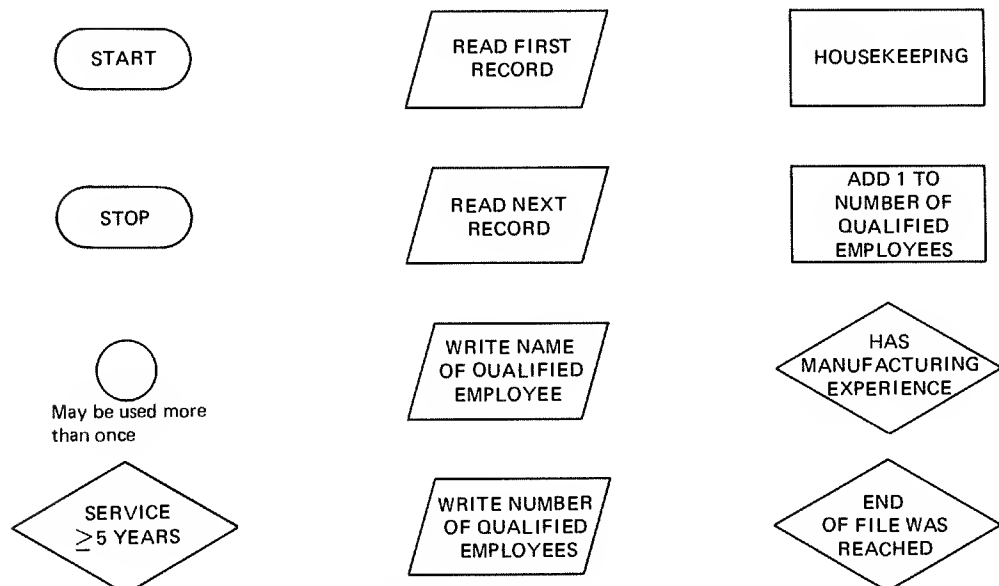


FIGURE 2.7 Scrambled flowchart

3. Figure 2.8 represents a COBOL program to process a file of student records and print the names of selected students. The selected students are to have at least 110 credits and also a major in engineering. As can be seen, various portions of the COBOL program have been blanked out. Restore the missing information so the program will run as intended.

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID. STUDENT.
000120 AUTHOR. MARION MILGROM.
000130
000140 ENVIRONMENT DIVISION.
000150 _____ 1
000160 SOURCE-COMPUTER. TRS-80.
000170 OBJECT-COMPUTER. TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210 SELECT _____ 2
000220 ASSIGN TO INPUT "STUDENT/DAT".
000230 SELECT PRINT-FILE
000240 ASSIGN TO PRINT "STUDENT/TXT".
000250
000260 DATA DIVISION.
000270 _____ 3
000280 FD STUDENT-FILE
000290 LABEL RECORDS ARE STANDARD
000300 RECORD CONTAINS _____ 4 CHARACTERS
000310 DATA RECORD IS STUDENT-RECORD.
000320 01 STUDENT-RECORD.
000330 05 STU-NAME PIC X(20).
000340 05 STU-MAJOR PIC X(10).
000350 05 STU-CREDITS PIC 9(3).
000360 05 FILLER PIC X(47).
000370
000380 _____ 5 PRINT-FILE
000390 LABEL RECORDS ARE STANDARD
000400 RECORD CONTAINS 132 CHARACTERS
000410 DATA RECORD IS PRINT-LINE.
000420 _____ 6 PRINT-LINE.
000430 05 FILLER PIC X(8).
000440 05 PRINT-NAME PIC X(25).
000450 05 FILLER PIC X(99).
000460
000470 WORKING-STORAGE SECTION.
000480 _____ 7 WS-DATA-REMAINS-SWITCH PIC X(3) VALUE SPACES.
000490
000500 PROCEDURE DIVISION.
000510 SELECT-ENGINEERING-SENIORS.
000520 OPEN _____ 8 STUDENT-FILE
000530 OUTPUT PRINT-FILE.
000540 MOVE SPACES TO PRINT-LINE.
000550 READ STUDENT-FILE
000560 AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
000570 PERFORM PROCESS-RECORDS
000580 UNTIL WS-DATA-REMAINS-SWITCH = "NO".
000590 _____ 9 STUDENT-FILE
000600 PRINT-FILE.
000610 STOP RUN.
000620
000630 _____ 10
000640 IF STU-MAJOR = "ENGINEERING" AND STU-CREDITS > 109
000650 MOVE SPACES TO PRINT-LINE
000660 MOVE STU-NAME TO PRINT-NAME
000670 WRITE PRINT-LINE
000680 AFTER ADVANCING 2 LINES.
000690
000700 READ _____ 11
000710 AT END MOVE "NO" TO _____ 12

```

FIGURE 2.8 Fill in the blanks

3

TRS-80 COBOL

OVERVIEW Chapters 1 and 2 presented rudiments of the COBOL language and programming logic. Attention was concentrated on COBOL as it applied to all computers, with no specific mention of the TRS-80. This chapter focuses on the TRS-80 Models II and III, with emphasis on the associated COBOL requirements.

We begin with a discussion of *machine* versus *problem* oriented languages with emphasis on the distinction between *compiling* and *executing* a program. The reader learns how to invoke the TRS-80 COBOL compiler, and to execute the generated object module.

The COBOL Source Program Editor, CEDIT, is introduced as the means of creating and modifying COBOL programs. All modes of CEDIT (insert, command, and edit) are covered, and the reader should become competent in its use. The PRINT utility program is presented to list the results of both compilation and execution. The concept of an operating system is introduced, and various TRSDOS (the TRS-80 Operating System) utilities and commands are highlighted. These include: FORMS, KILL, and DIR.

The chapter examines compiler output of the COBOL program, FIRST-TRY, which was introduced in Chapter 2. Attention is drawn to the various compiler options, e.g., a cross reference listing, as well as the COBOL requirements for entering information in specific columns.

It should be emphasized that this chapter deals with specifics of the TRS-80, and consequently is required reading for experienced COBOL programmers not familiar with this computer. The chapter presents a wealth of material associated with the TRS-80 operating system, but at an introductory level. The reader may also refer to the following manuals for greater coverage:

TRS-80 Model II (III) Disk Operating System Reference Manual
TRS-80 Model II (III) COBOL Language Manual
TRS-80 Model II (III) COBOL User's Guide

COBOL VERSUS MACHINE LANGUAGE

Every computer has its own unique machine language tied to specific locations in its memory. Human beings, however, think in terms of problems and use quantities with mnemonic significance, e.g., HOURS, RATE, PAY, etc. We might say that a person thinks in a problem oriented or higher level language while in actuality the computer functions in a machine oriented or lower level language. The two are related through a *compiler, which is a computer program that translates a problem (or source) language into a machine (or object) language*. COBOL is an example of a problem oriented language for business systems. The COBOL compiler is itself a machine language program written in the language of the machine on which it is executed.

The wide availability of COBOL compilers provides tremendous flexibility for individual programs. A COBOL program written for the TRS-80 can also execute on an IBM, Univac, Honeywell, NCR, or any other machine which has a COBOL compiler. The output from each compiler is different. The TRS-80 compiler produces a TRS-80 machine language program, a Univac compiler produces Univac language, etc. However, this does not concern the COBOL programmer. All he or she need know, and indeed care about, is COBOL: the compiler does the rest.

Consider a simple COBOL statement, MULTIPLY HOURS BY RATE

GIVING PAY. This takes HOURS, multiplies it by RATE, and puts the result into PAY. The values of HOURS and RATE are unchanged as a result of this instruction. For this instruction to execute, the compiler has to assign locations in its memory to HOURS, RATE, and PAY. It will multiply HOURS by RATE in a work area (known as an accumulator or register) and put the result into PAY.

Assume that the compiler decides to store HOURS, RATE, and PAY in locations 1000, 2000, and 3000, respectively. It then generates a sequence of three machine instructions to accomplish the intended multiplication:

```
LOAD      1000
MULTIPLY  2000
STORE     3000
```

The first instruction, LOAD 1000, brings the contents of location 1000 (HOURS) into the accumulator. The next instruction multiplies the contents of the accumulator by the contents of location 2000 (RATE). The result remains in the accumulator. Finally, the STORE instruction puts the contents of the accumulator into location 3000 (PAY). (Note that these instructions are typical of compilers in general and vary from computer to computer.)

Table 3.1 illustrates the results of three machine language instructions. It assumes values of 40 and 5 for HOURS and RATE, respectively. It shows the contents of locations 1000, 2000, 3000, and the accumulator before and after each of the three machine language instructions are executed. Prior to the LOAD instruction, the contents of both locations 3000 and the accumulator are immaterial. After the LOAD has been executed, the contents of location 1000 have been brought into the accumulator. After the MULTIPLY, the contents of the accumulator are 200, and after the STORE, the contents of location 3000 are also 200. Note that the initial contents of locations 1000 and 2000 are unchanged throughout.

TABLE 3.1
MACHINE INSTRUCTIONS TO MULTIPLY HOURS BY RATE

Instruction	Memory Contents							
	Before				After			
	1000 (HOURS)	2000 (RATE)	3000 (PAY)	ACCUM	1000 (HOURS)	2000 (RATE)	3000 (PAY)	ACCUM
LOAD 1000	40	5	?	?	40	5	?	40
MULTIPLY 2000	40	5	?	40	40	5	?	200
STORE 3000	40	5	?	200	40	5	200	200

A single COBOL statement invariably expands to one or more machine language statements after compilation. This phenomenon is known as *instruction explosion* and is a distinguishing characteristic of compiler languages. Compare the three machine language statements to the single COBOL statement. Obviously, the latter is shorter, but it is also easier to write, since the COBOL programmer need not remember which memory locations contain the data. In the early days of the computer age, there were no compilers, and all programs were written in machine language. Then someone had a remarkably simple yet powerful idea: Why not let the computer remember where the data are kept? The compiler concept was born, and things have never been the same since.

The remainder of the book deals exclusively with COBOL, rather than machine language. Indeed, there is no need for a competent COBOL pro-

programmer to even know machine language, provided one is aware of the critical role of the compiler. The programmer must, however, be knowledgeable about the computer's operating system.

THE TRS-80 OPERATING SYSTEM

An *operating system* is a set of machine language programs supplied by the manufacturer, which provide for the efficient operation of the computer. An operating system may include the COBOL compiler (remember a compiler is merely a computer program), compilers for other programming languages, and some general purpose programs known as utilities. The TRS-80 operating system is known as TRSDOS. Communication between the programmer and the operating system is accomplished through various commands, which are explained in this chapter. The operating system signals it is ready to accept a command with the message TRSDOS READY.

Many of the instructions to the operating system require the user to specify a file which may in turn be either a COBOL program, an operating system program, or simply a data file. The following conventions have been established with respect to file names:

1. A file is completely specified by a file name of up to eight characters, and a three letter extension, e.g., PAYROLL/CBL. (The extension, however, may sometimes be omitted, as when compiling or executing a program.)
2. Certain extensions have predetermined meanings, specifically:
 - CBL — applies to a COBOL source program and is *input* to the compiler
 - COB — denotes a COBOL object (machine language) program and is output from the compiler
 - LST — signifies a compiler listing which is also produced by the compiler
3. The *same file name often appears with different extensions*, e.g., PAYROLL/CBL and PAYROLL/COB.

Given this introduction to TRSDOS, we discuss how to utilize the TRS-80 operating system, how to compile programs, and how to execute the generated object programs.

COMPILATION ON THE TRS-80

TRSDOS signals it is ready to receive a command by the message, TRSDOS READY. Whenever this message appears, the COBOL compiler can be invoked by the command:

RSCOBOL filename options

where:

filename	is the name of a COBOL source program and is assumed to have the extension CBL, e.g., FIRSTTRY/CBL.
options	allow the user to control compiler output by invoking (or suppressing) various features. A partial set of compiler options is listed in Table 3.2. Multiple options must be separated by a space.

TABLE 3.2
SELECTED COMPILER OPTIONS

-
- L — Indicates that the compiler listing is to be written to a disk file with the same name as the COBOL source file, but with an extension of LST; e.g., FIRSTTRY/LST. The default is *not* to generate a listing.
 - O — Indicates that the output of the compiler, i.e., the object module is to be written to a disk file with the same name as the COBOL source file, but with an extension of COB; e.g., FIRSTTRY/COB. The default is to generate an object module. Specification of O=N will suppress the object file.
 - P — Indicates that the listing is to be printed. The default is *not* to print the listing. (Note, however, that one can print the listing at a later time through the PRINT utility program, which is explained in a subsequent section.)
 - T — Indicates the listing is to be displayed on the console. (The user can temporarily halt the listing by hitting the HOLD Key. HOLD must be hit a second time when the blinking cursor appears for the listing to resume.) The default is *not* to display the listing on the CRT.
 - X — Produces a cross-reference (alphabetical) listing of Data and Procedure Division names, indicating where each name is defined and referenced. Specification of this option requires that L, P, or T also be specified. The default is *not* to print a cross reference.
-

See COBOL User's Guide for additional information.

Consider the command, `RSCOBOL FIRSTTRY L X T`, to compile the source program `FIRSTTRY/CBL`. In addition, it will produce a cross-reference listing (X option), direct output to the terminal (T option), and put the compiler output into a disk file, `FIRSTTRY/LST` (L option). An object module, `FIRSTTRY/COB` is produced by default.

Compiler Output

Figure 3.1 is the compiler output which was produced by the command `RSCOBOL FIRSTTRY L X T`. Each page of output has identical heading information: date and time of compilation, compiler options, and file name as it appears on disk. (Note well that the heading's source file and PROGRAM-ID paragraph have the same entry, `FIRSTTRY`. This is the author's convention, rather than a system requirement, and simplifies keeping track of the many files on a diskette.)

The COBOL program is listed in its entirety. Observe that each statement has both a compiler line number, as well as a six-digit sequence number. (We learn later in the chapter that the sequence number is actually part of the COBOL statement and referenced by a text editor, CEDIT.)

Observe also that Procedure Division statements have an additional entry between the line and sequence numbers. These entries represent the address at which the generated machine language instructions for the COBOL statement begin. These addresses will prove useful in indicating where execution terminates.

The third page of compiler output in Figure 3.1 describes entries in the Data Division. Notice how *elementary* items are indented under their respective *group* item. (These terms are clarified in Chapter 4.) Note further how the length of every field is calculated.

The fourth page of output is a *cross-reference listing*, which has all programmer defined names in alphabetical order. The first number following the data name is the compiler statement where the item was defined. All subsequent numbers indicate where the data name is referenced or altered, e.g., `PRINT-SALARY` is defined in line 40 and altered in line 67. (Note that 0067 is enclosed in asterisks.) `EMP-NAME`, however, is merely referenced in line 65 as there are no asterisks around 0065.

File name as it appears in directory

TRS-B0 Model II COBOL (RM/COBOL 1.3B) 4/18/81 11.39.05 PAGE 1

SOURCE FILE: FIRSTTRY

Column 8

Column 12

OPTION LIST: L X T

Compiler options

Filename and extension of EMPLOYEE-FILE as it appears in directory

"FIRSTTRY/DAT".

"FIRSTTRY/TXT".

COBOL sequence numbers are in columns 1-6

Compiler statement number

```

LINE  DEBUG  PG/LN  A...B.....ID.....
1      000100  IDENTIFICATION DIVISION.
2      000110  PROGRAM-ID.  FIRSTTRY.
3      000120  AUTHOR.    MARION MILGROM.
4      000130
5      000140  ENVIRONMENT DIVISION.
6      000150  CONFIGURATION SECTION.
7      000160  SOURCE-COMPUTER.  TRS-B0.
8      000170  OBJECT-COMPUTER.  TRS-B0.
9      000180
10     000190  INPUT-OUTPUT SECTION.
11     000200  FILE-CONTROL.
12     000210      SELECT EMPLOYEE-FILE
13     000220          ASSIGN TO INPUT "FIRSTTRY/DAT".
14     000230      SELECT PRINT-FILE
15     000240          ASSIGN TO PRINT "FIRSTTRY/TXT".
16     000250
17     000260  DATA DIVISION.
18     000270  FILE SECTION.
19     000280  FD  EMPLOYEE-FILE
20     000290      LABEL RECORDS ARE STANDARD
21     000300      RECORD CONTAINS 80 CHARACTERS
22     000310      DATA RECORD IS EMPLOYEE-RECORD.
23     000320  01  EMPLOYEE-RECORD.
24     000330      05  EMP-NAME              PIC X(20).
25     000340      05  EMP-TITLE             PIC X(10).
26     000350      05  EMP-AGE              PIC 99.
27     000360      05  FILLER                PIC XX.
28     000370      05  EMP-SALARY            PIC 9(5).
29     000380      05  FILLER                PIC X(41).
30     000390
31     000400  FD  PRINT-FILE
32     000410      LABEL RECORDS ARE STANDARD
33     000420      RECORD CONTAINS 132 CHARACTERS
34     000430      DATA RECORD IS PRINT-LINE.
35     000440  01  PRINT-LINE.
36     000450      05  PRINT-NAME             PIC X(25).
37     000460      05  FILLER                 PIC XX.
38     000470      05  PRINT-AGE              PIC 99.
39     000480      05  FILLER                 PIC X(3).
40     000490      05  PRINT-SALARY            PIC 9(5).
41     000500      05  FILLER                 PIC X(95).
42     000510
43     000520  WORKING-STORAGE SECTION.
44     000530  77  WS-DATA-REMAINS-SWITCH    PIC X(3)      VALUE SPACES.
45     000540
46     000550  PROCEDURE DIVISION.
47     >0000  000560  SELECT-PROGRAMMERS.
48     >0000  000570      OPEN INPUT EMPLOYEE-FILE
49     000580          OUTPUT PRINT-FILE.
50     >0000  000590      MOVE SPACES TO PRINT-LINE.
51     >0010  000600      MOVE "SALRY REPORT FOR PROGRAMMERS UNDER 30" TO PRINT-LINE.
52     >0014  000610      WRITE PRINT-LINE
53     000620          AFTER ADVANCING 2 LINES.

```

TRS-B0 Model II COBOL (RM/COBOL 1.3B) 4/18/81 11.39.05 PAGE 2

SOURCE FILE: FIRSTTRY

A and B margins

OPTION LIST: L X T

Compilation page number

```

LINE  DEBUG  PG/LN  A...B.....ID.....
54     >001E  000630  READ EMPLOYEE-FILE
55     000640          AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
56     >0028  000650  PERFORM PROCESS-RECORDS
57     000660          UNTIL WS-DATA-REMAINS-SWITCH = "NO".
58     >0032  000670  CLOSE EMPLOYEE-FILE
59     000680          PRINT-FILE.

```

FIGURE 3.1 Compiler output

```

60 >003E 000690 STOP RUN.
61      000700 Hexadecimal address
62 >0040 000710 PROCESS-RECORDS.
63 >0040 000720 IF EMP-TITLE = "PROGRAMMER" AND EMP-AGE < 30
64      000730 MOVE SPACES TO PRINT-LINE
65      000740 MOVE EMP-NAME TO PRINT-NAME
66      000750 MOVE EMP-AGE TO PRINT-AGE
67      000760 MOVE EMP-SALARY TO PRINT-SALARY
68      000770 WRITE PRINT-LINE
69      000780 AFTER ADVANCING 2 LINES.
70      000790
71 >0066 000800 READ EMPLOYEE-FILE
72      000810 AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
73      ZZZZZZ END PROGRAM. *** END OF FILE ***

```

Last line is inserted by compiler.

TRS-80 Model 11 COBOL (RM/COBOL 1.3B) 4/18/81 11.39.05 PAGE 3
 SOURCE FILE: FIRSTTRY COBOL release OPTION LIST: L X T
 ADDRESS SIZE DEBUG ORDER TYPE NAM Date and time of compilation

>0000	80	GRP	0	FILE	EMPLOYEE-FILE
>0000	20	ANS	0	GROUP	EMPLOYEE-RECORD
>0014	10	ANS	0	ALPHANUMERIC	EMP-NAME
>001E	2	NSU	0	ALPHANUMERIC	EMP-TITLE
>0022	2	NSU	0	NUMERIC UNSIGNED	EMP-AGE
>0022	5	NSU	0	NUMERIC UNSIGNED	EMP-SALARY

>0054	132	GRP	0	FILE	PRINT-FILE
>0054	25	ANS	0	GROUP	PRINT-LINE
>006F	2	NSU	0	ALPHANUMERIC	PRINT-NAME
>0074	5	NSU	0	NUMERIC UNSIGNED	PRINT-AGE
>0074	5	NSU	0	NUMERIC UNSIGNED	PRINT-SALARY
>00E0	3	ANS	0	ALPHANUMERIC	WS-DATA-REMAINS-SWITCH

PRINT-SALARY is 5 characters

Elementary items are indented under group item

READ ONLY BYTE SIZE = >015B
 READ/WRITE BYTE SIZE = >015A
 OVERLAY SEGMENT BYTE SIZE = >0000
 TOTAL BYTE SIZE = >02B2

0 ERRORS
 0 WARNINGS

Compilation results

TRS-80 Model 11 COBOL (RM/COBOL 1.3B) 4/18/81 11.39.05 PAGE 4
 SOURCE FILE: FIRSTTRY OPTION LIST: L X T

CROSS REFERENCE /DECL/ *DEST

EMPLOYEE-FILE	/0012/ /0019/	0048	0054	0058	0071
EMPLOYEE-RECORD	/0023/				
EMP-AGE	/0026/ 0063	0066			
EMP-NAME	/0024/ 0065				
EMP-SALARY	/0028/ 0067				
EMP-TITLE	/0025/ 0063				
PRINT-AGE	/0038/ *0066*				
PRINT-FILE	/0014/ /0031/	0049	0059		
PRINT-LINE	/0035/ *0050*	*0051*	*0052*	*0064*	*0068*
PRINT-NAME	/0036/ *0065*				
PRINT-SALARY	/0040/ *0067*				
PROCESS-RECORDS	0056 /0062/				
SELECT-PROGRAMMERS	/0047/				
WS-DATA-REMAINS-SWITCH	/0044/ *0055*	0057	*0072*		

Data names appear alphabetically

PRINT-SALARY is altered in statement 67

PRINT-SALARY is defined in statement 40

FIGURE 3.1 Continued

EXECUTION ON THE TRS-80

Once compilation has been successfully completed, the programmer may attempt to execute the compiled program. This is accomplished by entering RUNCOBOL, followed by the filename; e.g., RUNCOBOL FIRSTTRY.

The relationship between compilation and execution is shown by Figure 3.2. Realize that *two* distinct programs are executed. In step 1, the compiler accepts FIRSTTRY/CBL as input and produces both a compiler listing FIRSTTRY/LST, and a machine language program, FIRSTTRY/COB as output. The latter program in turn is executed in step 2. It accepts FIRSTTRY/DAT (a data file as input) and produces the report FIRSTTRY/TXT as output.

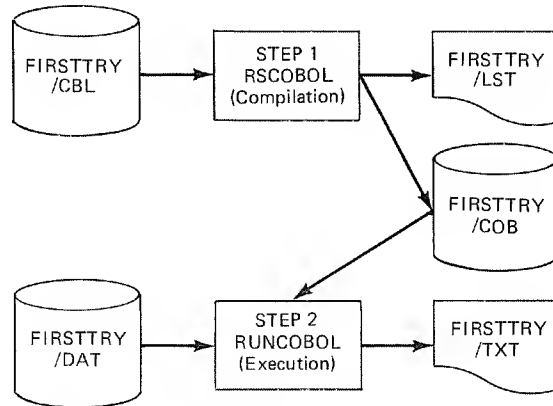


FIGURE 3.2 Compiling and executing a COBOL program

OBTAINING HARD COPY

The CRT is the primary output device of the TRS-80. It is ideal in many instances, but does not produce a permanent copy of the output. Very often it is necessary to obtain printed reports (hard copy) in lieu of, or in addition to, the terminal output. This is accomplished through the PRINT utility program.

Note, however, that the FORMS command must be issued before using PRINT. The FORMS utility is invoked simply by typing FORMS, and the utility will prompt the user to align paper prior to actual printing. Once FORMS has been executed successfully, the programmer can obtain hard copy of any file, by the command, PRINT filename/extension; for example, PRINT FIRSTTRY/LST.

DIR AND KILL COMMANDS

Any diskette is apt to contain many files, some of which we want to keep permanently and some of which are of temporary value. The command DIR lists *every* file on a diskette with its date of creation. Enter the command, DIR, and observe what happens. If more files exist than can be seen on a single screen, hit the HOLD key to stop the display. (HOLD must be hit a *second* time for the directory to continue.)

Very often, we will observe the presence of several files with the *same* name, but *different* extensions; e.g.:

FIRSTTRY/CBL — the COBOL source program

FIRSTTRY/COB — the object module

FIRSTTRY/LST — compiler listing

FIRSTTRY/TXT — output produced by executing the program FIRSTTRY/COB

FIRSTTRY/COB and FIRSTTRY/LST resulted from *compilation* of the COBOL source program, FIRSTTRY/CBL. FIRSTTRY/TXT is the output file produced by execution of the object program, FIRSTTRY/COB. (Check the SELECT statement of line 240 in Figure 3.1.) FIRSTTRY/CBL, and possibly FIRSTTRY/COB, are likely to be of *permanent* value, whereas the other files are not needed after printing. They may be deleted from the diskette through the KILL command, for example:

KILL FIRSTTRY/LST

The system will prompt with a message DELETE FIRSTTRY/LST (Y/N)?, verifying that we want to eliminate the file. *Note well that once a file is deleted, it is gone forever.* (The PURGE command will attempt to delete every file on a diskette, and will prompt the user accordingly. It is quite useful, especially after a long session on the machine.)

THE SOURCE PROGRAM EDITOR — CEDIT

All COBOL programs are created and modified through the program editor, CEDIT. CEDIT is a powerful tool and typical of editors on other computers. It is included on the COBOL diskette and is entered simply by typing CEDIT (after the message TRSDOS READY appears).

When called, the CEDIT program will announce itself, then end with a prompting character, > asking the user to input a command. Essentially, there are three modes of operation in CEDIT. These are:

INSERT	To create a COBOL program and/or to enter additional lines in an existing program. This mode is entered by typing I after the prompt character (>).
COMMAND	Enables the user to employ a variety of commands which include the ability to: delete an existing line or lines (D), find a particular character string (F), print one or more lines in the program (P), write a permanent copy of the program (W), replace an existing line (R), and enter the Insert (I) or Edit (E) modes.
EDIT	Allows complex editing of a <i>single line as opposed to</i> the command mode which looks at many lines within the program. The Edit Mode has several subcommands, which among other things, allow one to insert, replace, or delete characters within a single line.

Table 3.3 summarizes the Command Mode and is followed by a sample session, illustrating these commands. It must be emphasized, however, that this discussion is only an introduction to CEDIT. The reader is referred to the TRS-80 Model II (III) COBOL User's Guide for complete coverage.

TABLE 3.3
SUMMARY OF CEDIT COMMAND MODE

A	Automatic renumbering to avoid line collisions in the I or R commands.
B	Displays the first (beginning) line on the Model II, <i>but the bottom (last) line on the Model III</i> (This is one of the few differences between Model II and III editing commands).
C	Unconditional replacement (change) of one character string with another.
	C/OLD/NEW/ (Replaces the first occurrence of OLD with NEW)
	C/OLD/NEW/3 (Replaces the next three occurrences of OLD with NEW)
	C/OLD// (Replaces the next occurrence of OLD with nothing; i.e., it effectively deletes OLD from the line)
D	Deletes a line or lines.
	D 100 (Deletes line 100)
	D 100:200 (Deletes lines 100 through 200)
	D 5:* (Deletes line 5 through end of program)
E	Enters Edit Mode.
F	Finds a character string. Upon completion, the current line is set to the line of the last find.
	F/STRING/ (Finds the first occurrence of STRING)
	F/STRING/* (Finds all occurrences of STRING)
I	Enters the Insert Mode (to Escape, press either ESC or Break Keys).
	I 100, 1 (Begins inserting at line 100 with increments of 1)
L	Loads a CBL Source File
	L PAYROLL (Brings in the file PAYROLL/CBL as a work file)
N	Renumbers lines in the text.
	N 100, 20 (Renumbers all lines beginning at line 100 in increments of 20)
P	Prints the specified line or lines (if LINE-RANGE is omitted, the next 20 lines are printed).
	P 500 (Prints line 500)
	P 100:200 (Prints lines 100 to 200)
	P 200:* (Prints line 200 to end of program)
	P (Prints the next 20 lines)
Q	Quits CEDIT and returns to TRSDOS.
R	Replaces specified line and continues in Insert Mode (to exit Insert Mode, hit either ESC or Break Keys).
	R 100 (Prompts user to replace line 100)
S	Accepts a TRSDOS command and returns to CEDIT.
	S DIR (Displays directory of a diskette)
T	Displays the top line of a file (Model III only)
W	Writes current text (workfile) as a permanent file with extension CBL.
	W PAYROLL (Writes the file PAYROLL/CBL to the diskette)
X	Conditional replacement of one character string by another, i.e., change will not be made until user responds to prompt.
	X/OLD/NEW/ (Conditionally changes the next occurrence of OLD to NEW)
	X/OLD/NEW/3 (Conditionally changes the next three occurrences of OLD to NEW)
	X/OLD// (Conditionally removes the next occurrence of OLD)

A Sample Session

Figure 3.3 represents an initial (and rather poor) attempt at creating the pledge of allegiance. The many errors in Figure 3.3 are corrected in the session of Figure 3.4.

The editor is entered from the operating system through the CEDIT command. (All user responses are underlined; the system messages are not.) Next, the PLEDGE file is loaded into memory through the L command. Corrections may be entered in any order, and we begin by deleting line 110. The C (Change) command unconditionally changes the first occurrence of "AREMICA" to "AMERICA" and indicates the line in which the correction is made.

Line numbers

Line duplicated

Misspelling

```

00100 I PLEDGE ALLEGIANCE TO THE FLAG OF
00110 THE UNITED STATES OF AREMICA, AND TO
00120 THE UNITED STATES OF AREMICA, AND TO
00130 THEREPUBLIC FOR WHICH IT STANDS, ONE
00140 LIBERTY AND JUSTICE FOR ALL.

```

Space missing

Line missing-nation under God, . . .

FIGURE 3.3 Pledge of Allegiance (with errors)

We attempt to insert a space between “THE” and “REPUBLIC” with the X (conditional change) command, which finds the *first* occurrence of “THE” in line 120. Since this is *not* the desired change, we cancel it and rephrase the X command to change “THER” to “THE R”. This time it goes through as intended. This is an excellent example to illustrate the advantage of the conditional (X) over the unconditional (C) change command. The user

TRSDOS Message

```

TRSDOS READY
CEDIT
Tandem Systems Design
Cobol Compiler Editor Ver 2.0

```

Invokes CEDIT

```

Ready
:L PLEDGE
000195 Text
043729 Memory

```

Ready

```

:0110
C/AREMICA/AMERICA/
000120 THE UNITED STATES OF AMERICA, AND TO
:X/THE/THE /
000120 THE UNITED STATES OF AMERICA, AND TO
Change? N
:X/THER/THE R/
000130 THEREPUBLIC FOR WHICH IT STANDS, ONE
Change? Y
000130 THE REPUBLIC FOR WHICH IT STANDS, ONE
:1 135
000135 NATION, UNDER GOD, INDIVISIBLE, WITH
No Room Between Lines
:0
000100 I PLEDGE ALLEGIANCE TO THE FLAG OF
:N100,5
All Lines Re-numbered
:P
000100 I PLEDGE ALLEGIANCE TO THE FLAG OF
000105 THE UNITED STATES OF AMERICA, AND TO
000110 THE REPUBLIC FOR WHICH IT STANDS, ONE
000115 NATION, UNDER GOD, INDIVISIBLE, WITH
000120 LIBERTY AND JUSTICE FOR ALL.
:W NEWFILE

```

CEDIT Prompting character

User responses are underlined

X command produces a prompt before change is made

Exit from CEDIT

```

Ready
Q
TRSDOS READY

```

FIGURE 3.4 Sample session

is well advised to proceed with caution when substituting one character string for another.

The I (Insert) command is executed to add line 135, with the system responding NO ROOM BETWEEN LINES after the insertion. Recall that the default increment is 10, and hence no additional lines can be inserted before line 140. The problem can be circumvented by specifying a smaller increment, e.g., I 135, 1 or by the A (automatic renumbering) command.

The B (Beginning) command returns to the first line in the file and displays same to the user. (*Note well that on the Model III, B denotes bottom rather than beginning. The T command prints the top line.*) The N (Renumber) command renumbers all lines from 100, in increments of 5. The P (Print) command displays the file, W writes a new file to the disk, and Q (Quit) ends the session, returning control to the operating system.

It is suggested that the reader use the Source Program Editor and attempt to recreate the session of Figure 3.4. Enter CEDIT after the message TRSDOS READY appears. Create the file of Figure 3.3 through the INSERT command, I 100. Then follow Figure 3.4 exactly as it appears in the text.

CREATING A COBOL PROGRAM

A COBOL program is created by using CEDIT. It is critical, however, that various COBOL entries go in predetermined columns, as indicated by Table 3.4. CEDIT is quite helpful in insuring that the proper columns are used.

The insert mode automatically creates COBOL sequence numbers in columns 1 through 6, and positions the cursor in column 7. The TAB key spaces to column 8 (the A margin). Pressing TAB a second time positions the cursor to column 12 (the B margin). Every additional TAB will space four columns; i.e., to column 16, 20, 24, etc.

TABLE 3.4
COLUMN REQUIREMENTS FOR COBOL CODING

Columns	Description
1-6	COBOL sequence numbers which are automatically created by CEDIT.
7	Used to indicate comments and for continuation of non-numeric literals. A comment may appear anywhere in a COBOL program and is indicated by an * in column 7. Comments appear on the source listing, but are otherwise ignored by the compiler. Their use is encouraged to facilitate program documentation. Column 7 is also used to indicate continuation of non-numeric literals, and to control pagination of a source listing as explained in Chapter 7.
8-11	Known as the A margin. Division headers, section headers, paragraph names, FDs, 01s, and 77-level entries all begin in the A margin.
12-72	Known as the B margin. All remaining entries begin in or past column 12. COBOL permits considerable flexibility here, but individual installations have their own requirements. (See guidelines in Chapter 7.)
73-80	Program identification. A second optional field that is ignored by the compiler. Different installations have different standards, but the author suggests you omit these columns.

CREATING A DATA FILE

Unfortunately, CEDIT *cannot* be used to create a data file. The reader may wish to review Chapter 2 and/or Figure 3.2, with regard to the relationship between a program and the data on which it operates.

Figure 3.5 is the COBOL program which created the file, FIRSTTRY/DAT, the input to the program of Figure 3.1. The logic in Figure 3.5 is straightforward and should pose no problem in understanding. Realize that the file FIRSTTRY/DAT exists only after Figure 3.5 has been *compiled and*

SOURCE FILE: FIRSTDAT

OPTION LIST: T L=1

LINE	DEBUG	PG/LN	A...B.....	ID.....
1		000100	IDENTIFICATION DIVISION.	
2		000110	PROGRAM-ID. FIRSTDAT.	Filename on diskette and PROGRAM-ID match for convenience in identifying programs
3		000120	AUTHOR. R GRAUER.	
4		000130		
5		000140	ENVIRONMENT DIVISION.	Compiler statement number
6		000150	CONFIGURATION SECTION.	
7		000160	SOURCE-COMPUTER. TRS-80.	
8		000170	OBJECT-COMPUTER. TRS-80.	
9		000180		CEDIT line number
10		000190	INPUT-OUTPUT SECTION.	
11		000200	FILE-CONTROL.	
12		000210	SELECT EMPLOYEE-FILE	
13		000220	ASSIGN TO OUTPUT "FIRSTTRY/DAT".	
14		000230		
15		000240	DATA DIVISION.	Output of this program is input to program in Figure 3.1
16		000250	FILE SECTION.	
17		000260	FD EMPLOYEE-FILE	
18		000270	LABEL RECORDS ARE STANDARD	
19		000280	RECORD CONTAINS 80 CHARACTERS	
20		000290	DATA RECORD IS EMP-RECORD.	
21		000300	01 EMP-RECORD PIC X(80).	
22		000310		
23		000320	PROCEDURE DIVISION.	
24	>0000	000330	MAINLINE.	
25	>0000	000340	OPEN OUTPUT EMPLOYEE-FILE.	
26	>0006	000350	MOVE SPACES TO EMP-RECORD.	
27	>000A	000360	MOVE "JOHN DOE ANALYST 35 23000" TO EMP-RECORD.	
28	>000E	000370	WRITE EMP-RECORD.	
29	>001A	000380	MOVE "PEGGY WILCOX PROGRAMMER31 19000" TO EMP-RECORD.	
30	>001E	000390	WRITE EMP-RECORD.	
31	>002A	000400	MOVE "JOHN SMITH PROGRAMMER24 15000" TO EMP-RECORD.	
32	>002E	000410	WRITE EMP-RECORD.	
33	>003A	000420	MOVE "SHEILA LEVINE PROGRAMMER29 19000" TO EMP-RECORD.	
34	>003E	000430	WRITE EMP-RECORD.	
35	>004A	000440	MOVE "MARSHAL CRAWFORD MANAGER 33 28000" TO EMP-RECORD.	
36	>004E	000450	WRITE EMP-RECORD.	
37	>005A	000460	MOVE "STANLEY STEAMER PROGRAMMER22 39000" TO EMP-RECORD.	
38	>005E	000470	WRITE EMP-RECORD.	
39	>006A	000480	MOVE "BENJAMIN LEE PROGRAMMER26 12000" TO EMP-RECORD.	
40	>006E	000490	WRITE EMP-RECORD.	
41	>007A	000500	MOVE "DICK PERSNICKETY PROGRAMER 28 19000" TO EMP-RECORD.	
42	>007E	000510	WRITE EMP-RECORD.	
43	>008A	000520	MOVE "MARION MILGROM JR. PROG 24 10000" TO EMP-RECORD.	
44	>008E	000530	WRITE EMP-RECORD.	
45	>009A	000540	CLOSE EMPLOYEE-FILE.	
46	>00A0	000550	STOP RUN.	
47		000560		
48		ZZZZZZ	END PROGRAM.	*** END OF FILE ***

Employee records as they will appear in data file

FIGURE 3.5 Program to create data file

executed. Further, any changes to the data file require that the program of Figure 3.5 be modified, recompiled, and re-executed.

Figure 3.5 contains two sets of statement numbers. The left-most column contains the compiler statement number which need not be the same as the CEDIT line number; e.g., compiler statement 10 corresponds to CEDIT line 190. Note well that all CEDIT references *must* be to the CEDIT line number.

SUMMARY This is one of the longest, but most important, chapters in the entire book. It dealt with the TRS-80 rather than COBOL, because knowledge of the language, in and of itself, is insufficient to enable one to function as a pro-

grammer. It is also necessary to know specifics of the machine that one is using to obtain working programs.

The chapter opened with the definition of a compiler, and by contrasting machine and higher level languages. Specifics of compiling and executing a COBOL program were covered. The TRS-80 Disk Operating System, TRSDOS, was introduced as was the COBOL Source Program Editor, CEDIT. The procedure for creating a COBOL program was presented, with attention to COBOL coding requirements in specific columns. A distinction was drawn between a COBOL program and the data on which it operates.

TRUE/FALSE

1. RSCOBOL and RUNCOBOL are equivalent commands.
2. CBL is the extension for a source program.
3. A cross-reference listing is always provided with a COBOL compile.
4. The command X/PRT/PRINT/ *always* changes PRT to PRINT.
5. The command C/PRT/PRINT/10 changes PRT to PRINT in line 10.
6. The command RSCOBOL PAYROLL L produces the files PAYROLL/COB and PAYROLL/LST.
7. The command P 100:300 prints 20 lines.
8. Statement numbers can never be changed.
9. A COBOL program is required to create a data file.
10. The utility COBOLPRT is used to print a COBOL listing.

EXERCISES

1. Indicate the starting column (or columns) for each of the following:
 - (a) Division headers
 - (b) Comments
 - (c) Paragraph names
 - (d) Statements in the Procedure Division, except paragraph names
 - (e) WORKING-STORAGE SECTION
 - (f) 77-level entries
 - (g) 01-level entries
 - (h) 05-level entries
 - (i) PICTURE clauses
 - (j) OPEN statement
 - (k) WRITE statement
 - (l) SOURCE-COMPUTER
 - (m) SELECT statement
2. Match each item with its proper description

_____	(1) A margin
_____	(2) B margin
_____	(3) Comment
_____	(4) IDENTIFICATION DIVISION
_____	(5) PROCEDURE DIVISION
_____	(6) Hyphen
_____	(7) Non-numeric literal
_____	(8) Reserved word
_____	(9) Compiler
_____	(10) Literal

 - (a) Denoted by an asterisk in column 7
 - (b) First line in any COBOL program
 - (c) Often appears in data names
 - (d) Columns 12 through 72

- (e) Contains the logic of a program
 - (f) Limited to 120 characters and enclosed in quotes
 - (g) Where division, section, and paragraph headers begin
 - (h) Translates problem oriented language to machine oriented language
 - (i) Preassigned meaning
 - (j) A constant; may be numeric or non-numeric
3. Given the following sequence of machine language instructions,

```

LOAD      500
MULTIPLY  600
STORE     700
    
```

complete the following table of memory contents:

		Memory Contents							
		Before				After			
Instruction		500	600	700	ACC	500	600	700	ACC
LOAD	500	10	20	?	?	?	?	?	?
MULTIPLY	600	?	?	?	?	?	?	?	?
STORE	700	?	?	?	?	?	?	?	?

4. Assume that our hypothetical computer also has a machine language ADD instruction in addition to the LOAD, MULTIPLY, and STORE instructions described in the text. Specifically, "ADD X" will add the contents of location X to the contents of the accumulator and leave the sum in the accumulator. Show the series of machine language instructions which would probably be generated for the COBOL instruction "ADD A, B, C GIVING D". Assume A, B, C, and D are in locations 100, 200, 300, and 400, respectively.
5. Indicate the CEDIT instructions which are necessary to modify the program of Figure 3.5. Specifically:
- (a) Delete Marshal Crawford.
 - (b) Insert David Brown, a programmer earning \$18,000, and 26 years old.
 - (c) Change Marion Milgrom's age to 33 from 24.
6. Figure 3.6 contains an initial attempt at creating the Preamble to the United States Constitution. Indicate the necessary CEDIT instructions to correct *all* errors.

Duplicate lines

People missing

Space missing

```

00100 WE, THE OF THE UNITEDSTATES, IN ORDER TO
00110 FORM A MORE PERFECT ONION, ESTABLISH JUSTICE, INSURE
00120 DOMESTIC TRANQUILITY, PROVIDE FOR THE COMMON DEFENSE,
00121 DOMESTIC TRANQUILITY, PROVIDE FOR THE COMMON DEFENSE,
00130 PROMOTE THE GENERAL WELFARE, AND SECURE THE BLESSINGS
00140 OF LIBERTY TO OURSELVES AND OUR POSTERITY, DO ORDAIN
00150 AND ESTABLISH THIS CONSTITUTION FOR THE UNITEDSTATES
    
```

Wrong word

Extra space

Space missing

Last line missing-should be "of America"

FIGURE 3.6 Preamble text

PROJECTS

1. Develop a flowchart (or pseudocode) and a corresponding COBOL program for the EAST-WEST telephone company (serving 400 residents on an unknown island in the mid-Pacific). The object of the program is to process a customer file and indicate the payment due from each resident. (Residents come into the telephone office and pay their bill in person.) If column 42 of an incoming record contains an X, it means the customer is retired and pays only \$2.00. All others pay \$5.00. Incoming records are in the following format:

Field	Columns	Picture
LAST-NAME	1-15	X(15)
FIRST-NAME	16-30	X(15)
POST-OFFICE-BOX	31-34	9(4)
TELEPHONE-NUMBER	36-40	9(5)
RETIRED-INDICATOR	42	X

The printed output should contain all five fields from the input record in columns 2 to 16, 18 to 32, 41 to 44, 56 to 60, and 65, respectively. In addition, it should print the amount due in columns 70 to 74. In order to test your program, it will be necessary to create a *separate* data file; use the following test data:

MERKLE	RICHARD	0135 00025
OBRIEN	ROBERT	0625 00321
MERKLE	OLIVE	0330 00250 X
BLAKELY	BRIAN	0279 00639
KESSEL	SILVIA	0217 00433
SLY	CAREY	0934 00372
KARVAZY	KAREN	0666 00218 X
CRAWFORD	STACY	0555 00319 X
CRAWFORD	AMY	0567 00417

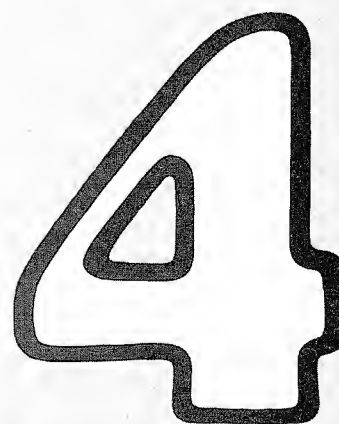
2. Develop pseudocode (or a flowchart) and a corresponding COBOL program for the Inter-City Piano Company. The program is to process a file of customers and produce a list of people eligible for a discount on buying a piano. Individuals with more than 15 lessons that have not already purchased a piano are eligible and should appear on the output. Individuals with 15 or fewer lessons or individuals who have already purchased a piano are not eligible. The format of the input records is as follows:

Field	Columns	Picture
LAST-NAME	1-15	X(15)
FIRST-NAME	16-25	X(10)
ADDRESS	26-50	X(25)
CITY	51-75	X(25)
NUMBER-OF-LESSONS	76-78	9(3)
PURCHASE-INDICATOR*	80	X

*Note: Y means already purchased.
N means not purchased.

The output print positions are 2 to 16, 18 to 27, 29 to 53, 55 to 79, 81 to 83, and 86 for the six incoming fields. No additional data need appear in the output, but remember *only* those customers eligible for a discount are to appear. Use the following test data and create a *separate* data file:

CRAWFORD	SHERRY	15004 GOOD MEADOW CT.	GAITHERSBURG	011 N
KARVAZY	KAREN	P. O. BOX 1013	GAITHERSBURG	017 Y
MORSE	KENNETH	11800 SILENT VALLEY LN.	GAITHERSBURG	014 N
PLUMETREE	MICHELE	14717 PEBBLE HILL RD.	GAITHERSBURG	027 N
SLY	MATTHEW	15001 GOOD MEADOW CT.	GAITHERSBURG	019 N
POWERS	NANCY	525 ORCHARD WAY	SILVER SPRINGS	024 Y
BLAKELY	KRISTEN	15005 ORCHARD WAY	SILVER SPRINGS	008 Y
BROWN	JENNIFER	11 HEATHER DRIVE	COLORADO SPRINGS	021 N
TARTLETON	KIMBERLY	BOX 395	NORTH LAPLATA	004 N

A large, stylized outline of the number 4, rendered in a thick black line. The number is positioned in the upper right quadrant of the page.

**THE COBOL
LANGUAGE**

OVERVIEW This chapter begins a formal treatment of COBOL by considering in order each of the four divisions. Emphasis, however, is on the Data and Procedure Divisions, which form the bulk of any COBOL program.

The material on the Identification and Environment Divisions is brief, but sufficient for elementary programs. Coverage of the Data Division begins with level numbers and picture clauses, and progresses to the File and Working-Storage Sections. The discussion on the Procedure Division includes verbs for I/O (OPEN, CLOSE, READ, and WRITE), for performing arithmetic (ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE), for implementing decisions and looping (IF and PERFORM), for transferring and editing data (MOVE), and finally for terminating a program (STOP RUN).

This unit contains a wealth of material. However, in keeping with the philosophy of the author (quick entry into actual programming), many of the discussions are kept brief. This chapter contains only necessary information for expanding on the programs of Chapters 1 and 2. Lengthy discussions are distinctly avoided. We could, for example, have devoted several pages to the IF and PERFORM verbs alone. We opted not to, in favor of encouraging quicker entry into actual programming.

We conclude with a complete COBOL program incorporating the major points of information. The program and associated discussion are extremely important. Accordingly, we suggest that as you read a section, skip over to the program and see how the material is applied. Again, do not be dismayed if you fail to remember everything on a first reading; rather, regard this unit as reference material to which you will continually return. ***Remember, the major portion of your learning will take place as you write your own programs.***

COBOL NOTATION COBOL is an English-like language in that there are a number of different but equally acceptable ways to say the same thing. Accordingly, a standard notation is used to concisely describe permissible COBOL formats. The notation takes a while to get used to, but once learned it permits the reader to quickly understand the syntax of any COBOL statement. The notation has six rules:

1. COBOL reserved words appear in uppercase (capital) letters.
2. Reserved words that are required are underlined; optional reserved words are not underlined.
3. Lowercase words denote programmer-supplied information.
4. Brackets ([]) indicate optional information.
5. Braces ({ }) indicate that one of the enclosed items must be chosen.
6. Three periods (...) mean that the last syntactical unit can be repeated an arbitrary number of times.

As illustration, variations in the IF statement are conveniently expressed through this notation. Consider:

$$\text{IF} \left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal-1} \end{array} \right\} \left\{ \begin{array}{l} \text{IS [NOT] GREATER THAN} \\ \text{IS [NOT] LESS THAN} \\ \text{IS [NOT] EQUAL TO} \end{array} \right\} \left\{ \begin{array}{l} \text{identifier-2} \\ \text{literal-2} \end{array} \right\}$$

Notice that the word IF is underlined and in uppercase letters; thus IF is a required reserved word. The first set of braces means that either a literal or identifier must appear; both are in lowercase letters, indicating that they are programmer supplied. The next set of braces forces a choice among one of three relationships: greater than, less than, or equal to. In each case, IS appears in capital letters but is not underlined; hence its use is optional. Brackets denote NOT as an optional entry. THAN is an optional reserved word, which may be added to improve legibility. Finally, a choice must be made between literal-2 or identifier-2.

Returning to the programmer selection problem of Chapter 2, in which we compared EMPLOYEE-TITLE to the literal "programmer", all the following are acceptable:

```
IF EMPLOYEE-TITLE IS EQUAL TO "PROGRAMMER" ...
IF EMPLOYEE-TITLE EQUAL "PROGRAMMER" ...
IF "PROGRAMMER" IS EQUAL TO EMPLOYEE-TITLE ...
```

This notation will be used throughout the chapter, and indeed throughout the book, to explain various COBOL elements.

IDENTIFICATION DIVISION

The Identification Division is the first of the four divisions in a COBOL program. Its function is to provide identifying information about the program, such as author, date written, security, and the like. The division consists of a division header and up to five paragraphs, as shown:

IDENTIFICATION DIVISION.

<u>PROGRAM-ID.</u>	program name.
[<u>AUTHOR.</u>	comment-entry.]
[<u>INSTALLATION.</u>	comment-entry.]
[<u>DATE-WRITTEN.</u>	comment-entry.]
[<u>SECURITY.</u>	comment-entry.]

Only the division header and PROGRAM-ID paragraph are required. (Note how brackets enclose the other paragraphs as per the COBOL notation.) Hence, the remaining paragraphs are optional and contain documentation about the program. (The DATE-COMPILED paragraph, which is supported on other compilers, is not available under TRS-80 COBOL.) A completed Identification Division is shown:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    TUITION.
AUTHOR.        ROBERT GRAUER.
INSTALLATION.  TANDY CORPORATION.
DATE-WRITTEN.  SEPTEMBER 1, 1983.
SECURITY.      TOP SECRET.
```

Coding for the Identification Division follows the general rules described in Chapter 3. The division header and paragraph names begin in the A margin. All other entries begin in or past column 12 (B margin).

ENVIRONMENT DIVISION

The Environment Division serves two functions:

1. It identifies the computer to be used for compiling and executing the program (usually one and the same). This is done in the Configuration Section.
2. It relates programmer chosen filenames to data files on a diskette. This is done in the Input-Output Section.

The nature of these functions makes the Environment Division heavily dependent on the computer on which one is working. Thus, the Environment Division for a COBOL program on the TRS-80 is significantly different from that of a program for another computer.

The Configuration Section has the format:

```
CONFIGURATION SECTION.  
SOURCE-COMPUTER.   TRS-80.  
OBJECT-COMPUTER.   TRS-80.
```

The section header and paragraph names begin in the A margin. The computer-name entries begin in or past column 12.

The Input-Output Section relates the files known to the COBOL program to the data files. Each file in a COBOL program has its own SELECT and ASSIGN clauses, which appear in the File-Control paragraph of the Input-Output Section of the Environment Division. The format of the ASSIGN clause is machine dependent, with the following entries typical for the TRS-80:

```
INPUT-OUTPUT SECTION.  
FILE-CONTROL.  
    SELECT EMPLOYEE-FILE ASSIGN TO INPUT "PAYROLL/DAT".  
    SELECT PRINT-FILE ASSIGN TO PRINT "PAYROLL/TXT".
```

EMPLOYEE-FILE is a programmer chosen file name and will appear elsewhere in the program (e.g., in FD, OPEN, CLOSE, and READ statements). It is designated as an input file, and exists on a diskette as PAYROLL/DAT. In similar fashion, PRINT-FILE is also a programmer chosen file name for a print (output) file which will exist on a diskette as PAYROLL/TXT.

DATA DIVISION

The Data Division describes all data names in a program as to: the number of characters, the type (for example, numeric or alphabetic), and, finally, the relationship among data items. This description of data is accomplished through the *picture* clause and *level* numbers.

Picture Clause

All data names are described according to size and class. *Size* specifies the number of characters in a field. *Class* denotes the type of field. For the present, we restrict type to alphabetic, numeric, or alphanumeric denoted by A, 9, or X, respectively. The size of a field is indicated by the number of times the A, 9, or X is repeated. Thus a data name with a picture of AAAA or A(4) is a four-position alphabetic field. In similar fashion, 999 and

X(5) denote a three-position numeric field and five-position alphanumeric field, respectively.

Level Numbers

Data items in COBOL are classified as either elementary or group items. A *group* item is one that can be further divided, whereas an *elementary* item cannot be further divided. Elementary items *always* have a picture clause, whereas group items *never* have a picture clause.

Level numbers are used to describe the hierarchy between group and elementary items. A level number of 01 indicates an entire record, whereas levels 02 through 49 are used for portions of a record.

Level numbers and picture clauses are best described by example. Consider the Data Division statements of Figure 4.1.

```

01 EMPLOYEE-RECORD.
   05 EMPLOYEE-NAME.
       10 LAST-NAME      PICTURE IS X(15).
       10 FIRST-NAME     PICTURE IS X(9).
       10 MIDDLE-INITIAL PICTURE IS X.
   05 EMPLOYEE-TITLE     PICTURE IS X(10).
   05 DATE-OF-BIRTH.
       10 BIRTH-MONTH    PICTURE IS 99.
       10 BIRTH-YEAR      PICTURE IS 99.
   05 PRESENT-SALARY      PICTURE IS 9(5).
   05 FILLER               PICTURE IS X(4).
   05 FORMER-SALARY       PICTURE IS 99999.

```

FIGURE 4.1 Data Division code for level numbers and PICTURE clause

EMPLOYEE-NAME is considered a group item since it is divided into three fields: LAST-NAME, FIRST-NAME, and MIDDLE-INITIAL. LAST-NAME, FIRST-NAME, and MIDDLE-INITIAL are elementary items since they are not further divided. EMPLOYEE-TITLE is an elementary item. DATE-OF-BIRTH is a group item divided into two elementary items, BIRTH-MONTH and BIRTH-YEAR. PRESENT-SALARY and FORMER-SALARY are both elementary items.

In Figure 4.1, EMPLOYEE-RECORD has a level number of 01. EMPLOYEE-NAME is a subfield of EMPLOYEE-RECORD, and hence it has a *higher* level number (05). LAST-NAME, FIRST-NAME, and MIDDLE-INITIAL are subfields of EMPLOYEE-NAME, and all have the level number 10. EMPLOYEE-TITLE, DATE-OF-BIRTH, PRESENT-SALARY, and FORMER-SALARY are also subfields of EMPLOYEE-RECORD and have the same level number as EMPLOYEE-NAME (05). DATE-OF-BIRTH in turn is subdivided into two elementary items, each with level number 10. *Realize that elementary items have a numerically higher level number than the group item to which they belong.*

Each elementary item must have a picture clause to describe the data it contains. LAST-NAME has PICTURE IS X(15), denoting a 15-position alphanumeric field. However, there is no picture entry for EMPLOYEE-NAME since that is a group item. Parentheses in a picture entry denote repetition; thus, the entries of 9(5) and 99999 for PRESENT-SALARY and FORMER-SALARY both depict five-position numeric fields. Finally, note the FILLER

entry of PICTURE IS X(4). FILLER denotes a field with no useful information, that is, data that are not referenced in this program.

Considerable flexibility is permitted with level numbers and picture clauses. Any level number from 02 through 49 is permitted in describing subfields as long as the basic rules are followed. Thus, 04 and 08 could be used in lieu of 05 and 10. Next, the picture clause itself can assume any one of four forms, PICTURE IS, PICTURE, PIC IS, or PIC. Finally, parentheses may be used to signal repetition of a picture type; A(3) is equivalent to AAA. Figure 4.2 is an alternate way of coding Figure 4.1, with emphasis on this flexibility.

```

01 EMPLOYEE-RECORD.
   04 EMPLOYEE-NAME.
       08 LAST-NAME          PIC X(15).
       08 FIRST-NAME         PIC X(9).
       08 MIDDLE-INITIAL     PIC X.
   04 EMPLOYEE-TITLE         PIC X(10).
   04 DATE-OF-BIRTH.
       08 BIRTH-MONTH        PIC 9(2).
       08 BIRTH-YEAR         PIC 9(2).
   04 PRESENT-SALARY         PIC 99999.
   04 FILLER                 PIC XXXX.
   04 FORMER-SALARY         PIC 99999.

```

FIGURE 4.2 Data Division code for level numbers and PICTURE clauses/II

File Section

The File Section is typically the first section in the Data Division. It describes every file mentioned in a SELECT statement in the Environment Division. (If, however, there are no input/output files as in the program of Chapter 1, there is no need for the File Section.)

The File Section contains both file description (FD) and record description entries (that is, level numbers and picture clauses). We have already discussed the latter. An abbreviated format for the file description (FD) entry is as follows:

```

FD file-name

  LABEL {RECORDS ARE} {OMITTED}
        {RECORD IS } {STANDARD}

  [RECORD CONTAINS integer-1 CHARACTERS]

  [DATA RECORD IS data-name-1]

```

The FD provides information about the physical characteristics of a file. The RECORD CONTAINS clause specifies the number of characters per record and should equal the sum of the picture clauses in the record description. The LABEL RECORDS clause indicates whether or not labels (i.e., predefined identifying information) exist for the file in question.

Working-Storage Section

The Working-Storage Section is used for storing intermediate results and/or constants needed by the program. It defines data names used by the program that are not defined elsewhere; i.e., in the File Section. Working-Storage

typically contains two types of entries. The first is for independent, elementary items; that is, those data names that have no hierarchical relationship to one another. These entries are assigned level number 77 and precede all other entries in Working-Storage. Group items beginning with level 01 are the second type of entry appearing in Working-Storage. Group items follow 77-level entries and use level numbers as discussed earlier. An example of a Working-Storage Section appears in Figure 4.3, which introduces the VALUE clause.

```

WORKING-STORAGE SECTION.
77  WS-DATA-REMAINS-SWITCH      PIC X(3)      VALUE SPACES.

01  HEADING-LINE.
    05  FILLER                  PIC X(8)      VALUE SPACES.
    05  FILLER                  PIC X(4)      VALUE "NAME".
    05  FILLER                  PIC X(9)      VALUE SPACES.
    05  FILLER                  PIC X(4)      VALUE "RATE".
    05  FILLER                  PIC X(4)      VALUE SPACES.
    05  FILLER                  PIC X(9)      VALUE "REG HOURS".
    05  FILLER                  PIC X(4)      VALUE SPACES.
    05  FILLER                  PIC X(9)      VALUE "O/T HOURS".
    05  FILLER                  PIC X(4)      VALUE SPACES.
    05  FILLER                  PIC X(7)      VALUE "REG PAY".
    05  FILLER                  PIC X(4)      VALUE SPACES.
    05  FILLER                  PIC X(7)      VALUE "O/T PAY".
    05  FILLER                  PIC X(4)      VALUE SPACES.
    05  FILLER                  PIC X(9)      VALUE "GROSS PAY".
    05  FILLER                  PIC X(46)     VALUE SPACES.

01  COMPANY-TOTALS.
    05  CO-REG-PAY              PIC 9(6)      VALUE ZEROS.
    05  CO-OVERTIME-PAY         PIC 9(6)      VALUE ZEROS.
    05  CO-GROSS-PAY            PIC 9(6)      VALUE ZEROS.

```

FIGURE 4.3 Working-Storage section

Value Clause

The VALUE clause is a convenient way of initializing a data name. It has the general form:

VALUE IS literal

Literals are of three types, numeric (for example, 30) non-numeric ("NAME"), and figurative constants (ZERO). Numeric and non-numeric literals were discussed in Chapter 2 as basic COBOL elements. Figurative constants are COBOL reserved words with preassigned values. Two of these, ZERO (equivalent forms are ZEROS and ZEROES) and SPACE (also SPACES), appear in Figure 4.3.

The Working-Storage Section of Figure 4.3 is extracted from the program at the end of the chapter. The exact nature of the various entries in Figure 4.3 will be better understood at that time.

PROCEDURE DIVISION

The Procedure Division is the portion of a COBOL program that contains its logic; it is the part of the program that “actually does something.” Coverage begins with the arithmetic verbs (ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE). We look at the READ, WRITE, OPEN, and CLOSE verbs for use in I/O operations. We study the MOVE verb, which transfers data from one area of memory to another. We learn basic forms of the IF and PERFORM statements and cover STOP RUN, to terminate execution. All of these verbs have a variety of options. This chapter, however, uses only the more elementary formats and defers additional coverage to later chapters.

ADD

The ADD verb has two basic formats:

$$\underline{\text{ADD}} \left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal-1} \end{array} \right\} \left[\begin{array}{l} \text{identifier-2} \\ \text{literal-2} \end{array} \right] \dots \underline{\text{TO}} \text{identifier-n}$$

and

$$\underline{\text{ADD}} \left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal-1} \end{array} \right\} \left\{ \begin{array}{l} \text{identifier-2} \\ \text{literal-2} \end{array} \right\} \left[\begin{array}{l} \text{identifier-3} \\ \text{literal-3} \end{array} \right] \dots \underline{\text{GIVING}} \text{identifier-n}$$

Note that one or several identifiers (literals) may precede identifier-n. Regardless of which format is chosen, i.e., GIVING or TO, only the value of identifier-n is changed. In the “TO” option, the values of identifier-1, identifier-2, etc., are added to the initial contents of identifier-n. In the “GIVING” option, the sum does not include the initial value of identifier-n. Simply stated, the “TO” option includes the initial value of identifier-n in the final sum, while the “GIVING” option ignores the initial value. Examples 4.1 and 4.2 illustrate both formats.

Example 4.1

ADD A B TO C.

Before execution: A

5

 B

10

 C

20

After execution: A

5

 B

10

 C

35

In Example 4.1 the initial values of A, B, and C are 5, 10, and 20, respectively. After execution the values are 5, 10, and 35. The instruction took the initial value of A (5), added the initial value of B (10), added the initial value of C (20), and put the sum (35) back into C.

Example 4.2

ADD A B GIVING C.

Before execution: A

5

 B

10

 C

20

After execution: A

5

 B

10

 C

15

In Example 4.2 the initial value of A (5) is added to the initial value of B (10), and the sum (15) replaces the initial value of C.

TABLE 4.1
THE ADD INSTRUCTION

Data name	A	B	C
Value <i>before</i> execution	5	10	30
Value <i>after</i> execution of			
ADD A TO C.	5	10	35
ADD A B TO C.	5	10	45
ADD A 18 B GIVING C.	5	10	33
ADD A 18 B TO C.	5	10	63
ADD 1 TO C.	5	10	31

Table 4.1 contains additional examples of the ADD instruction. In each instance, the instruction is assumed to operate on the initial values of A, B, and C (5, 10, and 30, respectively). Note that only the value of C changes.

SUBTRACT

The SUBTRACT verb also has two formats:

$$\text{SUBTRACT } \left\{ \begin{array}{c} \text{identifier-1} \\ \text{literal-1} \end{array} \right\} \left[\begin{array}{c} \text{identifier-2} \\ \text{literal-2} \end{array} \right] \dots \text{FROM identifier-m}$$

$$\text{SUBTRACT } \left\{ \begin{array}{c} \text{identifier-1} \\ \text{literal-1} \end{array} \right\} \left[\begin{array}{c} \text{identifier-2} \\ \text{literal-2} \end{array} \right] \dots \text{FROM } \left\{ \begin{array}{c} \text{identifier-m} \\ \text{literal-m} \end{array} \right\} \text{GIVING identifier-n}$$

In the first format, the initial value of identifier-m is replaced by the result of the subtraction. In the second format, the initial value of either identifier-m or literal-m is unchanged as the result is stored in identifier-n. Regardless of which option is used, the value of only one data name is changed. Consider examples 4.3 and 4.4.

Example 4.3

SUBTRACT A FROM B.

Before execution: A 5 B 15

After execution: A 5 B 10

In Example 4.3 the SUBTRACT verb causes the value of A (5) to be subtracted from the initial value of B (15) and the result (10) to be stored in B. Only the value of B was changed.

Example 4.4

SUBTRACT A FROM B GIVING C.

Before execution: A 5 B 15 C 100

After execution: A 5 B 15 C 10

In the “FROM . . . GIVING” format of Example 4.4 the value of A (5) is subtracted from the value of B (15), and the result (10) is placed in C. The

TABLE 4.2
THE SUBTRACT INSTRUCTION

Data name	A	B	C	D
Value <i>before</i> execution	5	10	30	100
Value <i>after</i> execution of				
SUBTRACT A FROM C.	5	10	25	100
SUBTRACT A B FROM C.	5	10	15	100
SUBTRACT A B FROM C GIVING D.	5	10	30	15
SUBTRACT 10 FROM C.	5	10	20	100

values of A and B are unchanged, and the initial value of C (100) is replaced by 10. Table 4.2 contains additional examples. In each example, the instruction is assumed to operate on the initial contents of A, B, and C.

MULTIPLY

The MULTIPLY format is shown below:

$$\underline{\text{MULTIPLY}} \left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal-1} \end{array} \right\} \underline{\text{BY}} \left\{ \begin{array}{l} \text{identifier-2} \\ \text{literal-2} \end{array} \right\} [\underline{\text{GIVING}} \text{ identifier-3}]$$

The use of GIVING is optional. If it is used, then the result of the multiplication is stored in identifier-3. If GIVING is omitted, then the result is stored in identifier-2. Either way, the value of only one data name is changed. Consider Examples 4.5 and 4.6.

Example 4.5

MULTIPLY A BY B.

Before execution: A 10 B 20

After execution: A 10 B 200

Example 4.6

MULTIPLY A BY B GIVING C.

Before execution: A 10 B 20 C 345

After execution: A 10 B 20 C 200

Table 4.3 contains additional examples of the MULTIPLY verb.

DIVIDE

The DIVIDE verb has two formats:

$$\underline{\text{DIVIDE}} \left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal-1} \end{array} \right\} \underline{\text{INTO}} \text{ identifier-2}$$

$$\underline{\text{DIVIDE}} \left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal-1} \end{array} \right\} \left\{ \begin{array}{l} \underline{\text{INTO}} \\ \underline{\text{BY}} \end{array} \right\} \left\{ \begin{array}{l} \text{identifier-2} \\ \text{literal-2} \end{array} \right\} \underline{\text{GIVING}} \text{ identifier-3}$$

TABLE 4.3
THE MULTIPLY INSTRUCTION

Data name	A	B	C
Value <i>before</i> execution	5	10	30
Value <i>after</i> execution of			
MULTIPLY B BY A GIVING C.	5	10	50
MULTIPLY A BY B GIVING C.	5	10	50
MULTIPLY A BY B.	5	50	30
MULTIPLY B BY A.	50	10	30
MULTIPLY A BY 3 GIVING C.	5	10	15

In the first format the quotient replaces the initial value of identifier-2. In the second format, the quotient replaces the initial value of identifier-3. In either case, the value of only one data name is changed. Consider Examples 4.7 and 4.8.

Example 4.7

DIVIDE A INTO B.

Before execution: A 10 B 50

After execution: A 10 B 5

Example 4.8

DIVIDE A INTO B GIVING C.

Before execution: A 10 B 50 C 13

After execution: A 10 B 50 C 5

In Example 4.7 the initial value of B (50) is divided by the value of A (10), and the quotient (5) replaces the initial value of B. In Example 4.8, which uses the “GIVING” option, the quotient goes into C and the values of A and B are unaffected.

Table 4.4 contains additional examples of the DIVIDE verb.

TABLE 4.4
THE DIVIDE INSTRUCTION

Data name	A	B	C
Value <i>before</i> execution	5	10	30
Value <i>after</i> execution of			
DIVIDE 2 INTO B.	5	5	30
DIVIDE 2 INTO B GIVING C.	5	10	5
DIVIDE B BY 5 GIVING A.	2	10	30
DIVIDE B INTO C.	5	10	3
DIVIDE A INTO B GIVING C.	5	10	2

COMPUTE

Any operation which can be done in an ADD, SUBTRACT, MULTIPLY, or DIVIDE statement may also be done using the COMPUTE instruction. In addition, the COMPUTE statement can combine different arithmetic operations in the same statement. For example, consider the following algebraic statement: $X = 2(A + B)/C$. A and B are first added together, the sum is multiplied by 2, and the product is divided by C. The single algebraic statement requires three COBOL arithmetic statements as shown. (Note that the true value of X is not obtained until after the last statement is executed.)

```
ADD A B GIVING X.
MULTIPLY 2 BY X.
DIVIDE C INTO X.
```

The previous statements can be combined into a single COMPUTE with obvious benefits:

```
COMPUTE X = 2 * (A + B) / C.
```

The general format of the COMPUTE statement is

```
COMPUTE identifier-1 = expression
```

Expressions are formed according to the following rules:

1. The symbols +, -, *, and / denote addition, subtraction, multiplication, and division, respectively. (Note well that exponentiation is *not* supported under TRS-80 COBOL.)
2. An expression consists of data names, literals, arithmetic symbols, and parentheses. Spaces must precede and follow arithmetic symbols.
3. Parentheses are used to *clarify* and in some cases *alter* the sequence of operations within a COMPUTE. Anything contained within the parentheses must also be a valid expression. The left parenthesis is preceded by a space, and the right parenthesis is followed by a space.

The COMPUTE statement calculates the value on the right side of the equal sign and stores it in the data name to the left of the equal sign. Expressions are evaluated as follows:

1. Anything contained in parentheses is evaluated first as a separate expression.

TABLE 4.5
THE COMPUTE INSTRUCTION

Data name	A	B	C	Comments
Value <i>before</i> execution	2	3	10	Initial values
Value <i>after</i> execution of COMPUTE C = A + B.	2	3	5	Simple addition
COMPUTE C = A + B * 2.	2	3	8	Multiplication done <i>before</i> addition
COMPUTE C = (A + B) * 2.	2	3	10	Parentheses evaluated first

2. Within the expression multiplication or division is done before addition or subtraction.
3. If rule 2 results in a tie, e.g., both addition and subtraction are present, then evaluation proceeds from left to right.

Table 4.5 contains examples to illustrate the formation and evaluation of expressions in a COMPUTE statement.

Table 4.6 should further clarify evaluation of the COBOL COMPUTE. This table contains several algebraic expressions and the corresponding COMPUTE statements to accomplish the intended logic. Note that parentheses are often required in the COMPUTE which are not present in the algebraic counterpart. Parentheses may also be optionally used to clarify the intent of a COMPUTE statement; however, their use in Table 4.6 is mandatory in all instances.

TABLE 4.6
THE COMPUTE INSTRUCTION CONTINUED

<i>Algebraic Expression</i>	<i>COBOL COMPUTE</i>
$x = a + b$	COMPUTE X = A + B.
$x = \frac{a + b}{2}$	COMPUTE X = (A + B) / 2.
$x = \frac{(a + b)c}{2}$	COMPUTE X = (A + B) * C / 2.
$x = \frac{a + b}{2c}$	COMPUTE X = (A + B) / (2 * C).

READ

The format for the READ verb is:

READ file-name [AT END imperative-statement]

As an example, consider:

```
READ EMPLOYEE-FILE
  AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
```

This statement causes a record to be read into memory. If, however, the end of file condition is reached (there are no more records), control passes to the statement following AT END, which moves "NO" to the data name WS-DATA-REMAINS-SWITCH.

WRITE

An abbreviated format of the WRITE verb is:

WRITE record-name $\left[\begin{Bmatrix} \text{AFTER} \\ \text{BEFORE} \end{Bmatrix} \right]$ ADVANCING $\left\{ \begin{matrix} \text{integer LINES} \\ \text{PAGE} \end{matrix} \right\}$

The WRITE statement transfers data from main storage to an output device. The ADVANCING option controls line spacing on a printer; if

omitted, single spacing occurs. If AFTER ADVANCING 3 LINES is used, the printer triple spaces (skips two lines and writes on the third). Output can be directed to a new page by specifying AFTER ADVANCING PAGE. The BEFORE option causes the line to be written first, after which the specified number of lines are skipped.

Note that the WRITE statement contains a *record* name, whereas the READ statement contains a *file* name. The record name in the WRITE will appear as an 01 entry in the File Section of the Data Division. The file name under which it is defined will appear in SELECT, FD, OPEN, and CLOSE statements.

OPEN

Every file in a COBOL program must be opened before it can be accessed. The OPEN verb causes the operating system to initiate action to make a file available for processing.

The format of the OPEN is:

$$\text{OPEN} \left\{ \begin{array}{l} \text{INPUT} \\ \text{OUTPUT} \end{array} \right. \text{file-name-1 [file-name-2 ...]} \right\}$$

Notice that one must specify the type of file in an OPEN statement. INPUT is used for a file that contains data, whereas OUTPUT is used for a file produced by a program; e.g., a printed report.

Two files may be opened in the same statement. For example,

```
OPEN INPUT EMPLOYEE-FILE
      OUTPUT PRINT-FILE.
```

CLOSE

All files must be closed before processing terminates. The format of the CLOSE is simply:

$$\text{CLOSE file-name-1 [file-name-2] ...}$$

Several files may be closed in the same statement. The type of file (INPUT or OUTPUT) is not specified. An example of a CLOSE statement follows:

```
CLOSE EMPLOYEE-FILE PRINT-FILE.
```

MOVE

The MOVE statement transfers data from one storage location to another. The format is:

$$\text{MOVE} \left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal} \end{array} \right\} \text{TO identifier-2}$$

Consider the examples:

```

MOVE 10000 TO STARTING-SALARY.
MOVE "SALARY REPORT FOR PROGRAMMERS UNDER 30" TO PRINT-LINE.
MOVE EMP-NAME TO PRINT-NAME.

```

The first example moves a numeric literal, 10000, to the data name STARTING-SALARY. The second moves a nonnumeric literal to PRINT-LINE. The third example transfers data from an input area to an output area for subsequent printing.

The figurative constants, ZEROS and SPACES, are frequently used in a MOVE, as shown:

```

MOVE SPACES TO PRINT-LINE.
MOVE ZEROS TO TOTAL-SALARIES.

```

The first statement moves spaces (blanks) to the data name PRINT-LINE. (Failure to clear a print-line in this fashion would result in the printing of any "garbage" that happened to be in these positions.) The second statement moves a numeric zero to TOTAL-SALARIES.

Although the MOVE statement appears rather elementary, care must be taken in its use. Certain moves, for example, are *not* permitted. One may not move a numeric field to an alphabetic field or vice versa. The MOVE statement may, however, contain fields of *different* lengths, e.g., a field with picture 9(3) moved to a field with picture 9(4) or a picture of X(6) moved to a picture of X(5), and so on.

Two additional rules are required, therefore, to clarify the action of the move:

1. Data moved from an alphanumeric area to an alphanumeric area are moved one character at a time from left to right. If the receiving field is larger than the sending field, it is padded on the right with blanks; if the receiving field is smaller than the sending field, the rightmost characters are truncated.
2. A numeric field moved to a numeric field is always aligned according to the decimal point. If the receiving field is smaller than the sending field, the high-order positions are truncated.

These rules are illustrated in Table 4.7.

TABLE 4.7
ILLUSTRATION OF THE MOVE STATEMENT

Source Field		Receiving Field	
PICTURE	CONTENTS	PICTURE	CONTENTS
X(5)	A B C D E	X(5)	A B C D E
X(5)	A B C D E	X(4)	A B C D
X(5)	A B C D E	X(6)	A B C D E
9(5)	1 2 3 4 5	9(5)	1 2 3 4 5
9(5)	1 2 3 4 5	9(4)	2 3 4 5
9(5)	1 2 3 4 5	9(6)	0 1 2 3 4 5

Editing Numeric Data

One of the major uses of the MOVE statement is in the editing of numeric data. *Incoming data are not allowed to contain decimal points, dollar signs, commas, or any other special characters. Printed reports, however, must contain these symbols to be of any use at all.* The conversion is accomplished through editing. Consider the two entries for FIELD-A and FIELD-A-EDITED:

```
05 FIELD-A          PIC 9V99.
05 FIELD-A-EDITED   PIC 9.99.
```

FIELD-A is a three-digit numeric field, with two digits after the decimal point. The V in its picture clause indicates an *implied* (or *assumed*) decimal point. FIELD-A-EDITED is a four-position edit field containing an actual decimal point. In a COBOL program, all calculations would be done using FIELD-A. Then, just prior to printing, FIELD-A is moved to FIELD-A-EDITED, and the latter field is printed. For example,

Before MOVE:

```

      V
FIELD-A 7 8 3  FIELD-A-EDITED ? . ? ?
```

After execution of MOVE FIELD-A TO FIELD-A-EDITED:

```

      V
FIELD-A 7 8 3  FIELD-A-EDITED 7 . 8 3
```

The decimal point takes an actual position in FIELD-A-EDITED, but does *not* occupy a position in FIELD-A, as it (the decimal point) is only implied. Although there are many editing symbols in COBOL, we discuss only the dollar sign, comma, and decimal point in this chapter and defer additional material to Chapter 8.

The appearance of a single \$ causes a dollar sign to print in the indicated position. Consider:

```
05 FIELD-B          PIC 9(3)V99.
05 FIELD-B-EDITED   PIC $(3).99.
```

Before MOVE:

```

      V
FIELD-B 6 5 4 3 2  FIELD-B-EDITED $ ? ? ? . ? ?
```

After execution of MOVE FIELD-B TO FIELD-B-EDITED:

```

      V
FIELD-B 6 5 4 3 2  FIELD-B-EDITED $ 6 5 4 . 3 2
```

Notice that the dollar sign and decimal point both take up a position in FIELD-B-EDITED.

It is also possible to obtain a *floating* dollar sign by using multiple dollar signs in the edited field. In this instance a single \$ prints immediately to the left of the most significant digit. Thus

```
05 FIELD-C          PIC 9(3)V99.
05 FIELD-C-EDITED   PIC $$$$.99.
```

Before MOVE:

				V					
FIELD-C	0	0	1	2	3		FIELD-C-EDITED	\$	\$

After execution of MOVE FIELD-C TO FIELD-C-EDITED:

				V					
FIELD-C	0	0	1	2	3		FIELD-C-EDITED		\$

A single dollar sign prints immediately before the left-most digit in the field. FIELD-C-EDITED is a seven-position field, but the first two positions hold blanks.

The presence of a comma as an editing symbol causes a comma to print if it is preceded by a significant digit. If, however, a comma is preceded only by zeros, then it is suppressed. Consider

```
05 FIELD-D          PIC 9(4).
05 FIELD-D-EDITED   PIC $$,$$9.
```

Before MOVE:

FIELD-D	8	7	6	5		FIELD-D-EDITED	\$	\$,	\$	\$?
---------	---	---	---	---	--	----------------	----	----	---	----	----	---

After execution of MOVE FIELD-D TO FIELD-D-EDITED:

FIELD-D	8	7	6	5		FIELD-D-EDITED	\$	8	,	7	6	5
---------	---	---	---	---	--	----------------	----	---	---	---	---	---

The comma prints in the indicated position. Suppose, however, that the contents of FIELD-D were 0087 instead of 8765. Now FIELD-D-EDITED would be:

Before MOVE:

FIELD-D	0	0	8	7		FIELD-D-EDITED	\$	\$,	\$	\$?
---------	---	---	---	---	--	----------------	----	----	---	----	----	---

After execution of MOVE FIELD-D TO FIELD-D-EDITED:

FIELD-D	0	0	8	7		FIELD-D-EDITED				\$	8	7
---------	---	---	---	---	--	----------------	--	--	--	----	---	---

Notice that the dollar sign floats and that the comma is suppressed. Note also that all numeric moves are accomplished so that decimal alignment is maintained with truncation or addition of insignificant zeros. Information on editing is summarized in Table 4.8.

TABLE 4.8
USE OF EDITING SYMBOLS

<i>Source Field</i>		<i>Receiving Field</i>	
<i>PICTURE</i>	<i>CONTENTS</i>	<i>PICTURE</i>	<i>CONTENTS</i>
9(4)	0678	9(4)	0678
9(4)	0678	\$9(4)	\$0678
9(4)	0678	\$\$\$\$	_\$678
9(4)V99	123456	9(4).99	1234.56
9(4)V99	123456	\$9(4).99	\$1234.56
9(4)V99	123456	\$9,999.99	\$1,234.56
9(4)	0008	,\$\$\$\$	____\$8
9(4)V9	12345	9(4)	1234
9(4)V9	12345	9(4).99	1234.50
99V99	1234	ZZZZ	\$__12

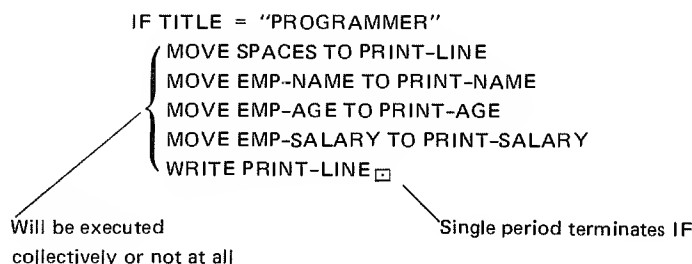
IF

The IF statement is used to implement a decision. For the present, our concern is only a few of the available options, and additional discussion is deferred to Chapter 6. The format of the IF is:

IF condition statement-1 [ELSE statement-2] .

The condition portion of the IF involves the comparison of two quantities. The syntax for specifying the condition was discussed in the COBOL notation section at the beginning of this chapter. As can be seen from the square brackets enclosing the ELSE clause, the IF statement may be used with or without the ELSE option.

Note well that either the word ELSE or a period terminates statement-1; that is, statement-1 (or for that matter statement-2) can contain several verbs. Consider:



If the condition is met, that is, if TITLE = "PROGRAMMER", then four distinct MOVES and one WRITE will be executed. In other words, it is the presence of the period that signifies the end of the IF. We return to this most important verb in Chapter 6.

PERFORM

The PERFORM verb is the primary means of implementing a loop. We discuss a simplified format in this section and defer additional material until Chapter 6. An abbreviated format is

PERFORM paragraph-name [UNTIL condition]

The PERFORM statement causes a portion of code to be executed. It transfers control to the paragraph specified, continues execution from that point until another paragraph is encountered, and then returns control to the statement immediately following the PERFORM statement. If the UNTIL clause is specified, the condition in the UNTIL clause is tested *prior* to any transfer of control. The performed paragraph is executed until the condition is met. In other words, when the condition is satisfied, the perform is finished, and control passes to the statement following the PERFORM. If, however, the condition is not satisfied, control is transferred to the designated paragraph.

Looping is accomplished by using the UNTIL clause, specifying a condition, and modifying that condition during execution of the performed paragraph. Consider:

```

77 END-OF-FILE-SWITCH    PIC X(2)    VALUE SPACES.
.
.
.
    PERFORM READ-A-RECORD
        UNTIL END-OF-FILE-SWITCH = "NO".
.
.
.
    READ-A-RECORD.
.
.
.
    READ EMPLOYEE-FILE
        AT END MOVE "NO" TO END-OF-FILE-SWITCH.

```

The paragraph READ-A-RECORD is performed until END-OF-FILE-SWITCH equals "NO", that is, until there are no more records. When the end of file is reached, the END-OF-FILE-SWITCH is set to NO. This causes the next test of the UNTIL condition to be met and prevents the READ-A-RECORD paragraph from further execution. Use of this technique to process a file also requires an *initial read*, as was shown in the programmer selection problem (Figure 2.5).

STOP RUN

Every program must contain at least one STOP RUN statement. When STOP RUN is encountered, execution of the COBOL program terminates and control passes back to the operating system.

It is important to realize that STOP RUN need not be the last physical statement in the Procedure Division. (It wasn't in Figure 2.5.) Rather, STOP RUN is said to indicate the logical end of the program, that is, the place where the programmer wants the job to end.

SUMMARY Chapters 1 and 2 began with almost immediate presentation of complete COBOL programs. The objective at that time was to remove the aura surrounding computer programming and to give the reader an intuitive feel for COBOL. This chapter formalized the COBOL presentation and introduced several new statements. Now we are ready to tie the material together and develop a more involved COBOL program. Specifications are as follows:

1. Develop a payroll program that will process a file of employee records and produce a printed report. The latter is to contain an ap-

appropriate heading at the beginning of the report, a detail line for every employee record, and a total line at the end.

2. Incoming records are formatted as follows:

<i>Field</i>	<i>Positions</i>	<i>Picture</i>
Employee Last Name	1-15	X(15)
Employee First Name	16-25	X(10)
Regular hours worked	26, 27	99
Overtime hours worked	28, 29	99
Hourly Rate	30-33	99V99

3. Processing specifications for each employee require calculation of regular, overtime, and gross pay. Regular pay is simply regular hours worked times the hourly rate; overtime pay is equal to overtime hours times the hourly rate times 1.5; gross pay is the sum of regular and overtime pay.
4. Company totals are required for regular, overtime, and gross pay.

The completed program is shown in Figure 4.4. Test data are shown in Figure 4.5 and the corresponding report in Figure 4.6. There are two SELECT statements in the COBOL program (lines 210-240). The first SELECT specifies that EMPLOYEE-FILE is to come from the data file, PAYROLL/DAT). The second SELECT statement shows that PRINT-FILE is written to the file PAYROLL/TXT. Realize, however, that PAYROLL/TXT will not exist on the diskette until *after* the program of Figure 4.4 compiles and executes. (The user can obtain "hard copy" through the PRINT utility as explained in Chapter 3.)

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID. PAYROLL.
000120 AUTHOR. R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER. TRS-80.
000170 OBJECT-COMPUTER. TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210     SELECT EMPLOYEE-FILE
000220     ASSIGN TO INPUT "PAYROLL/DAT".
000230     SELECT PRINT-FILE
000240     ASSIGN TO PRINT "PAYROLL/TXT".
000250
000260 DATA DIVISION.
000270 FILE SECTION.
000280 FD EMPLOYEE-FILE
000290 LABEL RECORDS ARE OMITTED
000300 RECORD CONTAINS 80 CHARACTERS
000310 DATA RECORD IS EMPLOYEE-RECORD.
000320 01 EMPLOYEE-RECORD.
000330    05 EMP-NAME.
000340       10 EMP-LAST-NAME PIC X(15).
000350       10 EMP-FIRST-NAME PIC X(10).

```

Input File as it exists on a diskette (See Figure 4.5)

Output File as it exists on a diskette (See Figure 4.6)

Only elementary items have picture clause

FIGURE 4.4 Payroll program

000360	05	EMP-HOURS-WORKED.		
000370	10	EMP-REG-HOURS	PIC 99.	Implied decimal point
000380	10	EMP-OVERTIME-HOURS	PIC 99.	
000390	05	EMP-RATE	PIC 99V99.	
000400	05	FILLER	PIC X(47).	
000410				
000420	FD	PRINT-FILE		A printer typically contains 132 print positions
000430		LABEL RECORDS ARE STANDARD		
000440		RECORD CONTAINS 132 CHARACTERS		
000450		DATA RECORD IS PRINT-LINE.		
000460	01	PRINT-LINE	PIC X(132).	
000470				
000480		WORKING-STORAGE SECTION.		
000490	77	WS-DATA-REMAINS-SWITCH	PIC X(3)	VALUE SPACES.
000500				
000510	01	IND-COMPUTATIONS.		
000520	05	IND-REGULAR-PAY	PIC 9(4)V99.	
000530	05	IND-OVERTIME-PAY	PIC 9(4)V99.	
000540	05	IND-GROSS-PAY	PIC 9(4)V99.	
000550				
000560	01	COMPANY-TOTALS.		Company totals are initialized to zero
000570	05	CO-REGULAR-PAY	PIC 9(6)V99	VALUE ZEROS.
000580	05	CO-OVERTIME-PAY	PIC 9(6)V99	VALUE ZEROS.
000590	05	CO-GROSS-PAY	PIC 9(6)V99	VALUE ZEROS.
000600				
000610	01	HEADING-LINE.		
000620	05	FILLER	PIC X(8)	VALUE SPACES.
000630	05	FILLER	PIC X(4)	VALUE "NAME".
000640	05	FILLER	PIC X(9)	VALUE SPACES.
000650	05	FILLER	PIC X(4)	VALUE "RATE".
000660	05	FILLER	PIC X(4)	VALUE SPACES.
000670	05	FILLER	PIC X(9)	VALUE "REG HOURS".
000680	05	FILLER	PIC X(4)	VALUE SPACES.
000690	05	FILLER	PIC X(9)	VALUE "O/T HOURS".
000700	05	FILLER	PIC X(4)	VALUE SPACES.
000710	05	FILLER	PIC X(7)	VALUE "REG PAY".
000720	05	FILLER	PIC X(4)	VALUE SPACES.
000730	05	FILLER	PIC X(7)	VALUE "O/T PAY".
000740	05	FILLER	PIC X(4)	VALUE SPACES.
000750	05	FILLER	PIC X(9)	VALUE "GROSS PAY".
000760	05	FILLER	PIC X(46)	VALUE SPACES.
000770				
000780	01	DASHED-LINE.		Heading line established through VALUE clauses
000790	05	ROW-OF-DASHES	PIC X(90)	VALUE ALL "-".
000800	05	FILLER	PIC X(42)	VALUE SPACES.
000810				
000820	01	DETAIL-LINE.		
000830	05	FILLER	PIC X(2).	
000840	05	DET-LAST-NAME	PIC X(15).	
000850	05	FILLER	PIC X(2).	
000860	05	DET-RATE	PIC \$\$\$.	99. Edit picture
000870	05	FILLER	PIC X(8).	
000880	05	DET-REG-HOURS	PIC Z9.	
000890	05	FILLER	PIC X(10).	
000900	05	DET-OVERTIME-HOURS	PIC Z9.	
000910	05	FILLER	PIC X(6).	
000920	05	DET-REGULAR-PAY	PIC \$Z,ZZ9.99.	
000930	05	FILLER	PIC X(3).	
000940	05	DET-OVERTIME-PAY	PIC \$Z,ZZ9.99.	
000950	05	FILLER	PIC X(2).	
000960	05	DET-GROSS-PAY	PIC \$Z,ZZ9.99.	
000970	05	FILLER	PIC X(47).	
000980				
000990	01	TOTAL-LINE.		
001000	05	FILLER	PIC X(6)	VALUE SPACES.
001010	05	FILLER	PIC X(6)	VALUE "TOTALS".
001020	05	FILLER	PIC X(41)	VALUE SPACES.

FIGURE 4.4 Continued

```

001030      05  TOTAL-REGULAR-PAY      PIC $Z,ZZ9.99.
001040      05  FILLER                PIC X(3)      VALUE SPACES.
001050      05  TOTAL-OVERTIME-PAY    PIC $Z,ZZ9.99.
001060      05  FILLER                PIC X(2)      VALUE SPACES.
001070      05  TOTAL-GROSS-PAY       PIC $Z,ZZ9.99.
001080      05  FILLER                PIC X(47)     VALUE SPACES.
001090
001100  PROCEDURE DIVISION.
001110  PREPARE-PAYROLL.
001120      OPEN INPUT EMPLOYEE-FILE
001130          OUTPUT PRINT-FILE.
001140      PERFORM WRITE-HEADING-LINE.
001150      READ EMPLOYEE-FILE
001160          AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
001170      PERFORM PROCESS-RECORDS
001180          UNTIL WS-DATA-REMAINS-SWITCH = "NO".
001190      PERFORM WRITE-COMPANY-TOTALS.
001200      CLOSE EMPLOYEE-FILE
001210          PRINT-FILE.
001220      STOP RUN.
001230
001240  WRITE-HEADING-LINE.
001250      WRITE PRINT-LINE FROM HEADING-LINE
001260          AFTER ADVANCING PAGE.
001270      WRITE PRINT-LINE FROM DASHED-LINE
001280          AFTER ADVANCING 1 LINE.
001290
001300  PROCESS-RECORDS.
001310      MULTIPLY EMP-REG-HOURS BY EMP-RATE GIVING IND-REGULAR-PAY.
001320      COMPUTE IND-OVERTIME-PAY
001330          = EMP-OVERTIME-HOURS * EMP-RATE * 1.5.
001340      ADD IND-REGULAR-PAY IND-OVERTIME-PAY GIVING IND-GROSS-PAY.
001350
001360      PERFORM UPDATE-COMPANY-TOTALS.
001370
001380      PERFORM WRITE-DETAIL-LINE.
001390
001400      READ EMPLOYEE-FILE
001410          AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
001420
001430  UPDATE-COMPANY-TOTALS.
001440      ADD IND-REGULAR-PAY TO CO-REGULAR-PAY.
001450      ADD IND-OVERTIME-PAY TO CO-OVERTIME-PAY.
001460      ADD IND-GROSS-PAY TO CO-GROSS-PAY.
001470
001480  WRITE-DETAIL-LINE.
001490      MOVE SPACES TO DETAIL-LINE.
001500      MOVE EMP-LAST-NAME TO DET-LAST-NAME.
001510      MOVE EMP-RATE TO DET-RATE.
001520      MOVE EMP-REG-HOURS TO DET-REG-HOURS.
001530      MOVE EMP-OVERTIME-HOURS TO DET-OVERTIME-HOURS.
001540      MOVE IND-REGULAR-PAY TO DET-REGULAR-PAY.
001550      MOVE IND-OVERTIME-PAY TO DET-OVERTIME-PAY.
001560      MOVE IND-GROSS-PAY TO DET-GROSS-PAY.
001570      WRITE PRINT-LINE FROM DETAIL-LINE
001580          AFTER ADVANCING 2 LINES.
001590
001600  WRITE-COMPANY-TOTALS.
001610      WRITE PRINT-LINE FROM DASHED-LINE
001620          AFTER ADVANCING 1 LINE.
001630      MOVE CO-REGULAR-PAY TO TOTAL-REGULAR-PAY.
001640      MOVE CO-OVERTIME-PAY TO TOTAL-OVERTIME-PAY.
001650      MOVE CO-GROSS-PAY TO TOTAL-GROSS-PAY.
001660      MOVE TOTAL-LINE TO PRINT-LINE.
001670      WRITE PRINT-LINE
001680          AFTER ADVANCING 2 LINES.

```

Use of editing

Initial read

Causes output to begin on a new page

Arithmetic Statements

Last statement of performed routine is another read

Increments company totals

Builds detail line

FIGURE 4.4 Continued

JONES	JOHN	40000500	EMP-RATE has Picture 99V99
DOBBS	JOHN	40080550	
BENJAMIN	LEE	30000675	
MILGROM	MARION	35100850	EMP-OVERTIME-HOURS
TATER	CRAIG	35080666	(positions 28, 29)
SMITH	JOHN	40100500	
DOE	JANE	40090550	EMP-REG-HOURS
			(positions 26, 27)

FIGURE 4.5 Input to Figure 4.4

The FD for EMPLOYEE-FILE describes the incoming records in accordance with the program's specifications, and contains both group and elementary items. Only the latter have picture clauses. The FD for PRINT-FILE states that print records contain 132 positions but the format of specific print lines is defined in Working-Storage.

The Procedure Division contains six paragraphs. The relationship among these paragraphs is best seen in the hierarchy chart of Figure 4.7. PREPARE-PAYROLL sits on top of the hierarchy chart and drives the entire program. It invokes three subordinate paragraphs, WRITE-HEADING-LINE, PROCESS-RECORDS, and WRITE-COMPANY-TOTALS. PROCESS-RECORDS in turn calls two other paragraphs which are subordinate to it.

The PREPARE-PAYROLL paragraph begins by opening the files (lines 1120 and 1130) and performing WRITE-HEADING-LINE. PAGE is used in the WRITE statement of line 1260 to cause output to begin on a new page. The WRITE FROM statement, lines 1270-1280, moves DASHED-LINE to PRINT-LINE and then writes PRINT-LINE. DASHED-LINE itself was previously defined in Working-Storage (lines 780-800) to consist of 90 dashes followed by 42 spaces. Realize that all "print lines" should add to 132 positions. In similar fashion, HEADING-LINE is defined in lines 610-760 and written in lines 1250 and 1260.

The READ statement of lines 1150 and 1160 obtains the *first* record in EMPLOYEE-FILE. The paragraph PROCESS-RECORDS is performed until no more records remain. Note well that the *last* statement of that routine

NAME	Incoming rate was edited			Fixed dollar sign produced by zero suppression			GROSS PAY
	RATE	REG HOURS	O/T HOURS	REG PAY	O/T PAY		
JONES	\$5.00	40	0	\$ 200.00	\$ 0.00	\$	200.00
DOBBS	\$5.50	40	8	\$ 220.00	\$ 66.00	\$	286.00
BENJAMIN	\$6.75	30	0	\$ 202.50	\$ 0.00	\$	202.50
MILGROM	\$8.50	35	10	\$ 297.50	\$ 127.50	\$	425.00
TATER	\$6.66	35	8	\$ 233.10	\$ 79.92	\$	313.02
SMITH	\$5.00	40	10	\$ 200.00	\$ 75.00	\$	275.00
DOE	\$5.50	40	9	\$ 220.00	\$ 74.25	\$	294.25
TOTALS				\$1,573.10	\$ 422.67		\$1,995.77

FIGURE 4.6 Output from Figure 4.4

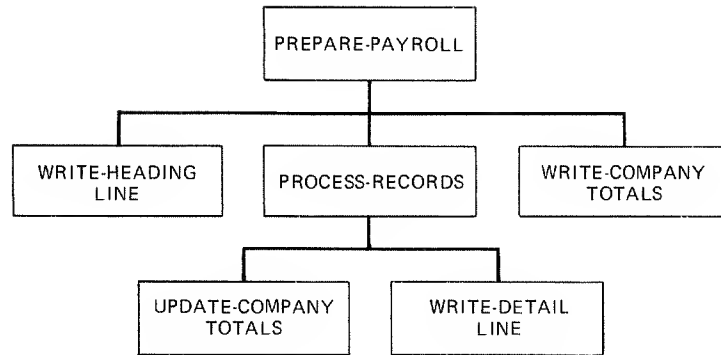


FIGURE 4.7 Hierarchy chart for payroll program

is another READ, in accordance with the guidelines of Chapter 2. After EMPLOYEE-FILE is empty, totals are written, the files are closed, and processing terminates.

Various arithmetic statements are used in PROCESS-RECORDS (lines 1310-1340) and also in the routine to increment company totals (lines 1440-1460). Note well that all arithmetic is performed on data names with strictly numeric pictures and *implied* decimal points; e.g., IND-OVERTIME-PAY in line 1320 with PIC 9(4)V99. However, edited fields with *actual* decimal points are printed; e.g., DET-OVERTIME-PAY with PIC \$Z,ZZ9.99 in line 940. The numeric field is moved to the edited field in line 1550 prior to writing the line. (The Z indicates zero suppression; i.e., do not print leading zeros as they are insignificant. Use of the Z produces a "fixed" dollar sign as can be seen in Figure 4.6.)

The reader should carefully examine the input data of Figure 4.5 and see how it is consistent with the report of Figure 4.6.

TRUE/FALSE

1. The PROGRAM-ID paragraph is the only required paragraph in the Identification Division.
2. Both GIVING and TO may appear in the *same* ADD statement.
3. All elementary items have a PICTURE clause.
4. A data name at the 10 level may or may not have a PICTURE clause.
5. PIC 9(3) and PICTURE IS 999 are equivalent entries.
6. The IF statement *must* contain an ELSE clause.
7. Only one statement is executed when an IF statement is true.
8. STOP RUN is physically the last statement of every COBOL program.
9. A file name may not appear in more than two statements in the same program.
10. In a COMPUTE statement with no parentheses, multiplication is always done before addition.

EXERCISES

1. Complete the following table. In each instance, refer to the *initial* values of A, B, C, and D.

Data name	A	B	C	D
Value <i>before</i> execution	4	8	12	1
Value <i>after</i> execution of				
ADD 1 TO D.				
ADD A B C GIVING D.				
ADD A B C TO D.				
SUBTRACT A B FROM C.				
SUBTRACT A B FROM C GIVING D.				
MULTIPLY A BY B.				
MULTIPLY B BY A.				
DIVIDE A INTO C.				
DIVIDE C BY A GIVING B.				
DIVIDE C BY B GIVING D.				
COMPUTE D = A + B / 2 * D.				
COMPUTE D = (A + B) / (2 * D).				
COMPUTE D = A + B / (2 * D).				
COMPUTE D = (A + B) / 2 * D.				
COMPUTE D = A + (B / 2) * D.				

2. Show the value of the edited result for each of the following entries:

Sending Field		Receiving Field	
PICTURE	CONTENTS	PICTURE	CONTENTS
(a) 9(6)	123456	9(6)	
(b) 9(6)	123456	9(8)	
(c) 9(6)	123456	9(6).99	
(d) 9(4)V99	123456	9(6)	
(e) 9(4)V99	123456	9(4)	
(f) 9(4)V99	123456	\$\$\$\$\$9.99	
(g) 9(4)V99	123456	\$\$\$,\$\$9.99	
(h) 9(6)	123456	\$\$\$\$,\$\$9.99	

3. Given the record layout for incoming data:

01	EMPLOYEE-RECORD.	
05	SOC-SEC-NUMBER	PICTURE IS 9(9).
05	EMPLOYEE-NAME.	
10	LAST-NAME	PICTURE IS X(12).
10	FIRST-NAME	PICTURE IS X(10).
10	MIDDLE-INIT	PICTURE IS X.
05	FILLER	PICTURE IS X.
05	BIRTH-DATE.	
10	BIRTH-MONTH	PICTURE IS 99.
10	BIRTH-DAY	PICTURE IS 99.
10	BIRTH-YEAR	PICTURE IS 99.
05	FILLER	PICTURE IS X(3).
05	EMPLOYEE-ADDRESS.	
10	NUMBER-AND-STREET.	
15	HOUSE-NUMBER	PICTURE IS X(6).
15	STREET-NAME	PICTURE IS X(10).
10	CITY-STATE-ZIP.	
15	CITY	PICTURE IS X(10).
15	STATE	PICTURE IS X(4).
15	ZIP	PICTURE IS 9(5).

- i. List all group items.
 - ii. List all elementary items.
 - iii. State the columns in which the following fields are found:
 - (a) SOC-SEC-NUMBER
 - (b) EMPLOYEE-NAME
 - (c) LAST-NAME
 - (d) FIRST-NAME
 - (e) MIDDLE-INIT
 - (f) BIRTH-DATE
 - (g) BIRTH-MONTH
 - (h) BIRTH-DAY
 - (i) BIRTH-YEAR
 - (j) EMPLOYEE-ADDRESS
 - (k) NUMBER-AND-STREET
 - (l) HOUSE-NUMBER
 - (m) STREET-NAME
 - (n) CITY-STATE-ZIP
 - (o) CITY
 - (p) STATE
 - (q) ZIP
4. Some of these statements are invalid. Indicate those that are and state why they are invalid. (Assume FILE-ONE and FILE-TWO are file names, and RECORD-ONE is a record name.)
- (a) OPEN INPUT RECORD-ONE.
 - (b) OPEN INPUT FILE-ONE OUTPUT FILE-TWO.
 - (c) OPEN INPUT FILE-ONE.
 - (d) CLOSE OUTPUT FILE-ONE.
 - (e) READ FILE-ONE.
 - (f) READ FILE-ONE AT END MOVE "YES" TO EOF-SWITCH.
 - (g) READ RECORD-ONE AT END MOVE "YES" TO EOF-SWITCH.
 - (h) WRITE RECORD-ONE.
 - (i) WRITE RECORD-ONE AFTER ADVANCING TWO LINES.
 - (j) WRITE RECORD-ONE BEFORE ADVANCING 2 LINES.
 - (k) CLOSE FILE-ONE FILE-TWO.
 - (l) WRITE FILE-ONE.
 - (m) WRITE RECORD-ONE AT END MOVE "YES" TO EOF-SWITCH.
 - (n) WRITE RECORD-ONE AFTER ADVANCING PAGE.
 - (o) WRITE RECORD-ONE AFTER 2.
 - (p) WRITE RECORD-ONE ADVANCING 3 LINES.
5. Write COBOL COMPUTE statements to accomplish the intended logic:
- (a) $x = a + b + c$
 - (b) $x = \frac{a + bc}{2}$
 - (c) $x = \frac{ab + cd}{ef}$
 - (d) $x = \frac{a + h}{2} - c$

PROJECTS

1. Write a program to itemize expenses for a building contractor. Each expenditure appears in a separate record according to the following format:

<i>Columns</i>	<i>Field</i>	<i>Picture</i>
1-8	DATE-OF-EXPENDITURE	X(8)
10	AUTHORIZATION-CODE	X
12-13	ITEM-TYPE	XX
15-39	ITEM-DESCRIPTION	X(25)
41-45	NUMBER-ORDERED	9(5)
47-55	UNIT-COST	9(7)V99

Print a heading line at the beginning of the report. Print a detail line for each incoming record that includes all incoming fields, as well as a calculated field equal to the NUMBER-ORDERED times the UNIT-COST. Accumulate a running total for cost, and print a total line at the end of processing. Use the following test data.

06/03/83	4	HD	ROOFING NAILS 15 LBS	00006	000000392
07/04/83	2	JL	GRAVEL-TRUCK LOAD	00042	000014000
06/05/83	4	AA	BATHROOM SINK-TYPE A	00094	000008000
06/20/83	2	JL	INDOOR LUMBER 2 x 3 x 10	02200	000000550
06/20/83	2	JL	CEMENT 50 LB BAGS	00320	000001570
07/15/83	3	FN	TRUSSES	00015	000012345
09/15/83	3	RF	ROOF TILES	04000	000000250

2. Write a program to print all shipments of furniture from a manufacturer to three different warehouses. A record is created every time a shipment is made, with the following format:

<i>Columns</i>	<i>Field</i>	<i>Picture</i>
1-6	DATE-OF-SHIPMENT	X(6)
7-13	ANTICIPATED-REVENUE	9(7)
14-15	TYPE-OF-SHIPMENT	XX
16	WAREHOUSE	X
23-52	PERSON-WHO-AUTHORIZED	X(30)

Design and print an appropriate heading line(s). Print a detail line for each incoming record. Keep a total of the anticipated-revenue for each warehouse (A, B, or C) and at the end of the run print three total lines, one for each warehouse. Sample test data are provided.

0212830030000ALC	RUSS FALLOWES
0214830070000NRC	RUSS FALLOWES
0219830002500NN8	DALE MANDRONA
0221830044000ALC	RAY DELODI
0228830010700RLA	PAUL ARON
0302830000200NCC	ART COOPER
0302830004600NN8	DALE MANDRONA
0309830004800NR8	DALE MANDRONA
0309830092000RLA	RAY DELODI

5

DEBUGGING

OVERVIEW Very few computer programs run successfully on the first attempt. Indeed, the programmer is realistically expected to make errors, and an important "test" of a good programmer is not whether he or she makes mistakes, but how quickly they are detected and corrected. Since this process is such an integral part of programming, an entire chapter is devoted to debugging. We shall consider errors in both compilation and execution.

Compilation errors occur in the translation of COBOL to machine language and result because the programmer has violated a rule of the COBOL grammar, e.g., a missing period, a misspelled word, or an entry in a wrong column. *Execution* errors result *after* the program has been successfully translated to machine language and are generally of two types:

1. The computer was able to execute the entire program, but the calculated results are different from that which the programmer expected or intended.
2. The computer is unable to execute a particular instruction and comes to a premature end of job, e.g., division by zero or addition of non-numeric data.

Execution errors of the first type may be caused by an incorrect translation of a proper flowchart or pseudocode to the programming language, or by a correct translation of an incorrect flowchart. In either case, there is an error in logic that is generating incorrect output. We shall restrict our discussion to compilation errors and execution errors of the first type.

ERRORS IN COMPILATION

A compilation error occurs because the COBOL syntax has not been followed. The compiler detects that something is amiss and responds with an error message. As we shall soon see, some of these messages are quite clear and consequently the problem is easily resolved. Others are not so precise, and require more time to correct.

The COBOL compiler tends to rub salt in a wound in the sense that an error in one statement can cause error messages in other statements that appear correct. For example, should you have an error in a SELECT statement, the compiler will flag the error, ignore the SELECT statement, and then flag any other statement which references that file even though those statements are otherwise correct.

Often simple mistakes such as omitting a statement or misspelling a reserved word can lead to a long and sometimes confusing set of error messages. The only consolation is that compiler errors can disappear as quickly as they occur. Correction of the misspelled word or insertion of the missing statement will often eliminate several errors at once.

Proficiency in debugging comes with time. The more programs you write, the better you become. To give you a truer feel for what to expect in your own programs, we have taken the payroll program of Chapter 4 and deliberately changed several of the statements to cause compilation errors. The resulting output is shown in Figure 5.1.

As a rule, the TRS-80 compiler lists errors immediately as they occur. Error messages usually appear in pairs—the first indicates the nature of the problem and the second the point at which compilation resumes. Consider statement 48 in Figure 5.1 in which the data name CO REGULAR-PAY is flagged because of a missing hyphen. The first message under statement 48 is SYNTAX, followed by a second message, SCAN RESUME. Each diagnostic


```
ILLEGAL PERFORM *E*E*E*E*E*E*E*E*E*E*E WRITE--COMPANY-TOTAL  
FILE RECORD SIZE ERROR *E*E*E*E*E*E*E*E*E EMPLOYEE-FILE  
VALUE ERROR *E*E*E*E*E*E*E*E*E*E*E TOTAL-LINE
```

4/18/81 12.05.04 PAGE 6
OPTION LIST: T L=1

14 ERRORS
11 WARNINGS

Compilation summary

FIGURE 5.1 *Continued*

is associated with a dollar sign indicating the column in line 48 to which the message pertains. The compiler detected an error when it encountered the R in REGULAR-PAY. Realize that CO was followed by a blank, causing CO to be construed as a data name. The compiler expected CO to be followed immediately by a picture clause, but encountered REGULAR-PAY instead. This in turn produced a syntax error, with the dollar sign appearing under the R in REGULAR-PAY.

Fortunately, the compiler does not give up completely when it encounters a syntax error. Instead, it seeks to recover, and resumes compilation as soon as it recognizes another valid element. Thus, when it detects the P in PIC in line 48, it prints a second message, SCAN RESUME, indicating that compilation is continuing. Accordingly, observe the second dollar sign under the P in PIC in line 48. The row of W's associated with the message, SCAN RESUME, denotes a warning, namely, that subsequent compilation may be incorrect.

With this as background, we are ready to consider the other diagnostics in Figure 5.1. The ensuing discussion references compiler line numbers, e.g., 48, rather than six-digit COBOL sequence numbers, e.g., 000570.

<i>Line Number</i>	<i>Explanation</i>
54	LITERAL VALUE—An opening quotation mark is missing before the literal NAME. Compilation resumes with the ending period.
58	SYNTAX—A somewhat perplexing error in that beginning and ending quotation marks appear to be present. Recall, however, that the compiler interprets only the first 72 columns, and that columns 73 through 80 are ignored. This problem occurred because the ending quotation mark is in column 73. This error is rather subtle, but no one ever said that programming was easy. Note, however, the column indicators for the A and B margins, as well as the program ID (columns 73–80) at the top of each page of a compiler output.
59	DOUBLE DECLARATION—Another puzzling error which occurred only because of the previous problem. Since the period in line 58 was in column 74, that entry was never terminated. Hence, when the scan continued at the picture clause in line 59, the compiler thought it had two pictures for a single entry; hence, the error and DOUBLE DECLARATION message.
102	RESERVED WORD CONFLICT—Recall that COBOL has a series of some 300 reserved words which can only be used in rigidly defined context. The error message indicates a reserved word conflict, meaning that START is unsuitable as a paragraph name (check Appendix B, Reserved Words). The error is

- eliminated by choosing another paragraph name, e.g., START-THE-PROGRAM.
- 116 REFERENCE INVALID—In COBOL, one reads a file, but writes a record. PRINT-FILE is a file name, rather than a record name; hence, the error. The problem is corrected by specifying PRINT-LINE rather than PRINT-FILE.
- 131 FILE NAME REQUIRED—Similar to the previous error, except that line 131 is a READ statement. The correct reference is EMPLOYEE-FILE, not EMPLOYEE-RECORD.
- 132 RESERVED WORD CONFLICT—The compiler resumed scanning with the word MOVE and immediately flagged NO as a reserved word conflict. (Verify that NO is indeed a reserved word in Appendix B.) The problem is eliminated by enclosing NO in quotes as was done in line 107.
- 135, 154 IDENTIFIER—The compiler is flagging CO-REGULAR-PAY as an invalid identifier because it was never defined in the Data Division. We attempted to define it in line 48, but omitted the hyphen, and the compiler is not a mind reader. These errors will disappear with the earlier correction to line 48.
- 137 IDENTIFIER—Similar in concept to the previous message in that IND-GROSS was never defined, although IND-GROSS-PAY was specified in line 45. We know the two data names refer to the same quantity, but the compiler requires identical data names and flags the inconsistency.

As was stated earlier, the compiler flags most errors immediately as they occur. Nevertheless, it may list additional errors at the conclusion of a compilation following the Data Division expansion. Consider now the three diagnostics on the bottom of the fifth page of compiler output in Figure 5.1.

ILLEGAL PERFORM: WRITE-COMPANY-TOTAL

There is nothing syntactically wrong with the PERFORM statement of line 110. Unfortunately, the paragraph WRITE-COMPANY-TOTAL does not exist and consequently the statement was flagged. Note that there is a paragraph, WRITE-COMPANY-TOTALS, but the compiler requires *exact* correspondence. One must be absolutely sure that paragraph names in a PERFORM statement match exactly the paragraph names as they appear elsewhere in the program.

FILE RECORD SIZE ERROR: EMPLOYEE-FILE

The RECORD CONTAINS clause of line 21 specified that EMPLOYEE-FILE will have 80-character records, but the PICTURE clauses in lines 24

VALUE ERROR: TOTAL-LINE

The results of the compilation are summarized in the very last item of compiler output. This particular example had 14 errors and 11 warnings.

The point of this example is that a seemingly simple error in one statement can lead to a large number of errors in other related statements that disappear when the initial error is corrected. However, don't expect all errors you don't understand to just go away. There is always a logical explanation for everything a computer does, and sometimes it may take quite a while to find it.

[illegible]

FIGURE 5.2 COBOL listing with *environment* misspelled

[illegible]

TRS-80 Model II COBOL (RM/COBOL 1.3B)
SOURCE FILE: PAYCOMP2

```
LINE  DEBUG PG/LN  A...B.....ID.....
```

FIGURE 5.2 Continued


```

LINE  DEBUG PG/LN  A...B.....ID.....

```

[illegible]

TRS-80 Model II COBOL (RM/COBOL 1.3B)
SOURCE FILE: PAYCOMP2

4/18/81 12.10.06 PAGE 6
OPTION LIST: T L=1

ADDRESS	SIZE	DEBUG	ORDER	TYPE	NAM
>0004	3	ANS	0	ALPHANUMERIC	WS--DATA--REMAINS--SWITCH
>0008	18	GRP	0	GROUP	IND--COMPUTATIONS
>0008	6	NSU	0	NUMERIC UNSIGNED	IND--REGULAR--PAY
>000E	6	NSU	0	NUMERIC UNSIGNED	IND--OVERTIME--PAY
>0014	6	NSU	0	NUMERIC UNSIGNED	IND--GROSS--PAY
>001A	24	GRP	0	GROUP	COMPANY--TOTALS
>001A	8	NSU	0	NUMERIC UNSIGNED	CO--REGULAR--PAY
>0022	8	NSU	0	NUMERIC UNSIGNED	CO--OVERTIME--PAY
>002A	8	NSU	0	NUMERIC UNSIGNED	CO--GROSS--PAY
>0032	132	GRP	0	GROUP	HEADING--LINE
>00B6	132	GRP	0	GROUP	DASHED--LINE
>00B6	90	ANS	0	ALPHANUMERIC	ROW--OF--DASHES
>013A	132	GRP	0	GROUP	DETAIL--LINE
>013C	15	ANS	0	ALPHANUMERIC	DET--LAST--NAME
>014D	6	NSE	0	NUMERIC EDITED	DET--RATE
>015B	2	NSE	0	NUMERIC EDITED	DET--REG--HOURS
>0167	2	NSE	0	NUMERIC EDITED	DET--OVERTIME--HOURS
>016F	9	NSE	0	NUMERIC EDITED	DET--REGULAR--PAY
>017B	9	NSE	0	NUMERIC EDITED	DET--OVERTIME--PAY
>0186	9	NSE	0	NUMERIC EDITED	DET--GROSS--PAY
>01BE	132	GRP	0	GROUP	TOTAL--LINE
>01F3	9	NSE	0	NUMERIC EDITED	TOTAL--REGULAR--PAY
>01FF	9	NSE	0	NUMERIC EDITED	TOTAL--OVERTIME--PAY
>020A	9	NSE	0	NUMERIC EDITED	TOTAL--GROSS--PAY

READ ONLY BYTE SIZE = >0132

READ/WRITE BYTE SIZE = >024C

OVERLAY SEGMENT BYTE SIZE = >0000

TOTAL BYTE SIZE = 0037E

20 ERRORS
19 WARNINGS

39 errors result because of one mistake

FIGURE 5.2 Continued

ERRORS IN EXECUTION

After a program has successfully compiled, it proceeds to execute and therein lies the strength and weakness of the computer. The primary attractiveness of the machine is its ability to perform a fantastic number of operations in infinitesimal amounts of time; its weakness stems from the fact that it does exactly what it has been instructed to do. The machine cannot think for itself. The programmer must think for the machine. If you were inadvertently to instruct the computer to compute pay by *adding* hours and rate, then that is what it would do.

To give you an idea of what can happen, we have deliberately altered the original payroll problem of Chapter 4 and created a new program. That, in turn, created the output of Figure 5.3, which at first glance resembles the original output of Figure 4.6. There are, however, subtle errors as follows:

1. Regular pay is not totalled for the company.
2. The total gross pay equals the gross pay of the last employee, and is obviously wrong.
3. Cents are missing from regular pay, overtime pay, and gross pay.
4. Overtime pay is calculated incorrectly.
5. The last record (Doe) is processed twice.
6. The first record (Jones) is missing.

First record (Jones) is missing				Overtime pay should be \$66.00			
NAME	RATE	REG HOURS	O/T HOURS	REG PAY	O/T PAY	GROSS PAY	
DOBBS	\$5.50	40	8	\$ 220.00	\$ 44.00	\$ 264.00	
BENJAMIN	\$6.75	30	0	\$ 202.00	\$ 0.00	\$ 202.00	
MILGROM	\$8.50	35	10	\$ 297.00	\$ 85.00	\$ 382.00	
FATER	\$6.66	35	8	\$ 233.00	\$ 53.00	\$ 286.00	
SMITH	\$5.00	40	10	\$ 200.00	\$ 50.00	\$ 250.00	
DOE	\$5.50	40	9	\$ 220.00	\$ 49.00	\$ 269.00	
DOE	\$5.50	40	9	\$ 220.00	\$ 49.00	\$ 269.00	
TOTALS				\$ 0.00	\$ 330.00	\$ 269.00	
Last record processed twice				Cents are missing			
				Regular pay is not totaled		Gross pay total is wrong	

FIGURE 5.3 Incorrect payroll report (produced by Figure 5.4)

Note well that Figure 5.4, the program that produced the output of Figure 5.3, compiled cleanly, with no error messages. Put another way, the errors inherent in Figure 5.3 are errors in execution, rather than compilation. The compiler successfully translated the COBOL program of Figure 5.4 into machine language because it (the program) was syntactically correct. Unfortunately, the program was logically incorrect, and hence the errors in Figure 5.3. Each error is discussed in detail.

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID. PAYLOGIC.
000120 AUTHOR. R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER. TRS-80.
000170 OBJECT-COMPUTER. TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210     SELECT EMPLOYEE-FILE
000220         ASSIGN TO INPUT "PAYROLL/DAT".
000230     SELECT PRINT-FILE
000240         ASSIGN TO PRINT "PAYLOGIC/TXT".
000250
000260 DATA DIVISION.
000270 FILE SECTION.
000280 FD EMPLOYEE-FILE
000290     LABEL RECORDS ARE OMITTED
000300     RECORD CONTAINS 80 CHARACTERS
000310     DATA RECORD IS EMPLOYEE-RECORD.
000320 01 EMPLOYEE-RECORD.
000330     05 EMP-NAME.
000340         10 EMP-LAST-NAME          PIC X(15).
000350         10 EMP-FIRST-NAME         PIC X(10).
000360     05 EMP-HOURS-WORKED.
000370         10 EMP-REG-HOURS          PIC 99.
000380         10 EMP-OVERTIME-HOURS     PIC 99.
000390     05 EMP-RATE                   PIC 99V99.
000400     05 FILLER                     PIC X(47).
000410
000420 FD PRINT-FILE
000430     LABEL RECORDS ARE STANDARD
000440     RECORD CONTAINS 132 CHARACTERS
000450     DATA RECORD IS PRINT-LINE.
000460 01 PRINT-LINE                     PIC X(132).
000470
000480 WORKING-STORAGE SECTION.
000490 77 WS-DATA-REMAINS-SWITCH        PIC X(3)      VALUE SPACES.
000500
000510 01 IND-COMPUTATIONS.
000520     05 IND-REGULAR-PAY            PIC 9(4).
000530     05 IND-OVERTIME-PAY          PIC 9(4).
000540     05 IND-GROSS-PAY             PIC 9(4).
000550
000560 01 COMPANY-TOTALS.
000570     05 CO-REGULAR-PAY            PIC 9(6)V99  VALUE ZEROS.
000580     05 CO-OVERTIME-PAY          PIC 9(6)V99  VALUE ZEROS.
000590     05 CO-GROSS-PAY             PIC 9(6)V99  VALUE ZEROS.
000600
000610 01 HEADING-LINE.
000620     05 FILLER                     PIC X(8)      VALUE SPACES.
000630     05 FILLER                     PIC X(4)      VALUE "NAME".
000640     05 FILLER                     PIC X(9)      VALUE SPACES.
000650     05 FILLER                     PIC X(4)      VALUE "RATE".
000660     05 FILLER                     PIC X(4)      VALUE SPACES.
000670     05 FILLER                     PIC X(9)      VALUE "REG HOURS".
000680     05 FILLER                     PIC X(4)      VALUE SPACES.
000690     05 FILLER                     PIC X(9)      VALUE "O/T HOURS".
000700     05 FILLER                     PIC X(4)      VALUE SPACES.
000710     05 FILLER                     PIC X(7)      VALUE "REG PAY".
000720     05 FILLER                     PIC X(4)      VALUE SPACES.
000730     05 FILLER                     PIC X(7)      VALUE "O/T PAY".
000740     05 FILLER                     PIC X(4)      VALUE SPACES.
000750     05 FILLER                     PIC X(9)      VALUE "GROSS PAY".
000760     05 FILLER                     PIC X(46)     VALUE SPACES.
000770
000780 01 DASHED-LINE.
000790     05 ROW-OF-DASHES             PIC X(90)     VALUE ALL "-".

```

Pictures should include implied decimal points

FIGURE 5.4 Payroll program with logic errors

```

000800      05  FILLER                                PIC X(42)    VALUE SPACES.
000810
000820 01  DETAIL-LINE.
000830      05  FILLER                                PIC X(2).
000840      05  DET-LAST-NAME                          PIC X(15).
000850      05  FILLER                                PIC X(2).
000860      05  DET-RATE                              PIC $$$99.
000870      05  FILLER                                PIC X(8).
000880      05  DET-REG-HOURS                          PIC Z9.
000890      05  FILLER                                PIC X(10).
000900      05  DET-OVERTIME-HOURS                     PIC Z9.
000910      05  FILLER                                PIC X(6).
000920      05  DET-REGULAR-PAY                       PIC $Z,ZZ9.99.
000930      05  FILLER                                PIC X(3).
000940      05  DET-OVERTIME-PAY                       PIC $Z,ZZ9.99.
000950      05  FILLER                                PIC X(2).
000960      05  DET-GROSS-PAY                         PIC $Z,ZZ9.99.
000970      05  FILLER                                PIC X(47).
000980
000990 01  TOTAL-LINE.
001000      05  FILLER                                PIC X(6)    VALUE SPACES.
001010      05  FILLER                                PIC X(6)    VALUE "TOTALS".
001020      05  FILLER                                PIC X(41)   VALUE SPACES.
001030      05  TOTAL-REGULAR-PAY                     PIC $Z,ZZ9.99.
001040      05  FILLER                                PIC X(3)    VALUE SPACES.
001050      05  TOTAL-OVERTIME-PAY                    PIC $Z,ZZ9.99.
001060      05  FILLER                                PIC X(2)    VALUE SPACES.
001070      05  TOTAL-GROSS-PAY                       PIC $Z,ZZ9.99.
001080      05  FILLER                                PIC X(47)   VALUE SPACES.
001090
001100  PROCEDURE DIVISION.
001110  PREPARE-PAYROLL.
001120      OPEN INPUT EMPLOYEE-FILE
001130          OUTPUT PRINT-FILE.
001140      PERFORM WRITE-HEADING-LINE.
001150      READ EMPLOYEE-FILE
001160          AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
001170      PERFORM PROCESS-RECORDS
001180          UNTIL WS-DATA-REMAINS-SWITCH = "NO".
001190      PERFORM WRITE-COMPANY-TOTALS.
001200      CLOSE EMPLOYEE-FILE
001210          PRINT-FILE.
001220      STOP RUN.
001230
001240  WRITE-HEADING-LINE.
001250      WRITE PRINT-LINE FROM HEADING-LINE
001260          AFTER ADVANCING PAGE.
001270      WRITE PRINT-LINE FROM DASHED-LINE
001280          AFTER ADVANCING 1 LINE.
001290
001300  PROCESS-RECORDS.
001310      READ EMPLOYEE-FILE
001320          AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
001330      MULTIPLY EMP-REG-HOURS BY EMP-RATE GIVING IND-REGULAR-PAY.
001340      COMPUTE IND-OVERTIME-PAY = EMP-OVERTIME-HOURS * EMP-RATE * 1.5.
001350      ADD IND-REGULAR-PAY IND-OVERTIME-PAY GIVING IND-GROSS-PAY.
001360
001370      PERFORM UPDATE-COMPANY-TOTALS.
001380
001390      PERFORM WRITE-DETAIL-LINE.
001400
001410
001420  UPDATE-COMPANY-TOTALS.
001430      ADD IND-OVERTIME-PAY TO CO-OVERTIME-PAY.
001440      ADD IND-GROSS-PAY TO CO-GROSS-PAY.
001450
001460  WRITE-DETAIL-LINE.
001470      MOVE SPACES TO DETAIL-LINE.
001480      MOVE EMP-LAST-NAME TO DET-LAST-NAME.
001490      MOVE EMP-RATE TO DET-RATE.
001500      MOVE EMP-REG-HOURS TO DET-REG-HOURS.
001510      MOVE EMP-OVERTIME-HOURS TO DET-OVERTIME-HOURS.

```

READ should be last statement of performed routine

5 is in column 73, and ignored by compiler

CO-REGULAR-PAY is not incremented

FIGURE 5.4 Continued

```

001520      MOVE IND-REGULAR-PAY TO DET-REGULAR-PAY.
001530      MOVE IND-OVERTIME-PAY TO DET-OVERTIME-PAY.
001540      MOVE IND-GROSS-PAY TO DET-GROSS-PAY.
001550      WRITE PRINT-LINE FROM DETAIL-LINE
001560          AFTER ADVANCING 2 LINES.
001570
001580      WRITE-COMPANY-TOTALS.
001590          WRITE PRINT-LINE FROM DASHED-LINE
001600              AFTER ADVANCING 1 LINE.
001610      MOVE CO-REGULAR-PAY TO TOTAL-REGULAR-PAY.
001620      MOVE CO-OVERTIME-PAY TO TOTAL-OVERTIME-PAY.
001630      MOVE IND-GROSS-PAY TO TOTAL-GROSS-PAY.
001640      MOVE TOTAL-LINE TO PRINT-LINE.
001650      WRITE PRINT-LINE
001660          AFTER ADVANCING 2 LINES.

```

Wrong field moved to total line

FIGURE 5.4 *Continued*

1. *Regular pay is not totalled:* The function of the paragraph, UPDATE-COMPANY-TOTALS is to increment all fields for which a running total is required. That was done successfully for both overtime and gross pay. It was not done for regular pay because the statement ADD IND-REG-PAY TO CO-REG-PAY is missing.
2. *Total gross pay is wrong:* CO-GROSS-PAY is incremented correctly in line 1440, so that the problem must be elsewhere. The paragraph WRITE-COMPANY-TOTALS builds a total line, then writes it. The error is in line 1630, in which IND-GROSS-PAY rather than CO-GROSS-PAY is moved to TOTAL-GROSS-PAY.
3. *Cents missing:* DET-REGULAR-PAY, defined in line 920 with PIC \$Z,ZZ9.99 contains the printed value of regular pay. IND-REGULAR-PAY, defined in line 520 with PIC 9(4), holds the calculated value for regular pay as determined by the MULTIPLY statement in line 1330. The problem arises because of an inconsistency in the two picture clauses. We are printing a field with a decimal point, *without* having calculated the decimal value. IND-REGULAR-PAY has a picture of 9(4), but should have had a picture of 9(4)V99 instead. The compiler was not told to retain a decimal value for IND-REGULAR-PAY, so it didn't; the fact that this field is subsequently moved to one containing a decimal point is immaterial.
4. *Overtime pay is calculated incorrectly:* This is a very subtle error, and one the author almost missed. Overtime pay is simply time and a half for each overtime hour. The COMPUTE statement of line 1340 appears correct, yet the calculated value is wrong. Dobbs, for example, worked 8 overtime hours. His regular pay rate is \$5.50 per hour, so his overtime pay should be \$66 ($8 \times \5.50×1.5) rather than \$44. The \$44 for Dobbs represents straight time, i.e., $8 \times \$5.50$, rather than time and a half. It's as if the .5 were missing from the COMPUTE statement of line 1340, yet .5 appears on the listing. This apparent contradiction is resolved when we realize *the 5 is in column 73!* In effect, the COMPUTE statement ended with the period after the 1 (in column 72), ignoring the entries in columns 73 and 74.
5. *Last record processed twice:* Recall that when the skeletal COBOL outline was presented in Chapter 2, there was an initial READ statement in the mainline paragraph and a second READ as the *last* statement in the performed routine. That structure is correct. In Figure 5.4, the initial READ is still present, but the second READ was *incorrectly* moved to the beginning of the performed routine, causing the last record to be processed twice. Eliminate the problem by moving lines 1310 and 1320 immediately after line 1390.

6. *First record missing:* This is a secondary effect of the previous error. The first record Jones is correctly read by the initial READ of lines 1150 and 1160. However, since the first statement of the performed routine is also a READ, Jones will be immediately replaced by Dobbs before any processing takes place. This problem disappears with the previous correction.

It is important to emphasize that these execution errors are not contrived, but typical of students and beginning programmers. Even the accomplished practitioner can be guilty of similar errors when rushed or careless. Realize also that execution errors occur without fanfare. There are no compiler diagnostics to warn of impending trouble. The program has compiled cleanly, and if it goes to a normal end of job, there is nothing to indicate a problem.

SUMMARY Don't be discouraged or surprised if you have many compilation errors in your first few attempts. Remember that a single error in a COBOL program can result in many error messages and that the errors can be made to disappear in bunches. (Recall what happened when environment was misspelled in Figure 5.2.)

Before leaving the subject of compilation errors, it is worthwhile to review a list of common errors and suggested ways to avoid them.

1. *Nonunique data names:* Occurs because the same data name is defined in two different records or twice within the same record. For example, NAME might be specified as input data in an EMPLOYEE-FILE and printed as output in a PRINT-FILE. To avoid the problem of nonunique data names, it is best to prefix every data name within a file by a short prefix. EMP- could be established as a prefix for EMPLOYEE-FILE, and PRINT as the prefix for PRINT-FILE as shown. This also helps locate data names while writing and debugging programs.

```

      .
      .
FD  EMPLOYEE-FILE
      .
      .
      DATA RECORD IS EMP-RECORD.
01  EMP-RECORD.
      05  EMP-NAME             PIC X(20).
      05  EMP-SOC-SEC-NO      PIC 9(9).
      .
      .
FD  PRINT-FILE
      .
      .
      DATA RECORD IS PRINT-RECORD.
01  PRINT-RECORD.
      05  PRINT-NAME          PIC X(20).
      05  FILLER              PIC X(5).
      05  PRINT-SOC-SEC-NO    PIC 9(9).

```

2. **Omitted periods:** Every COBOL sentence should have a period and omission usually results in the compiler's assumption of a period. Realize, however, that the period has special significance with respect to the IF statement where its omission (or inclusion) does not cause compiler errors, but significantly alters the program's logic.
3. **Omitted space before/after an arithmetic operator:** The arithmetic operators *, /, +, and - all require a blank before and after them. (A typical error for FORTRAN or PL/1 programmers since the space is not required in those languages.)
4. **Invalid picture for numeric entry:** All data names used in arithmetic statements must have numeric pictures. Permissible entries include: 9's, and a V.
5. **Conflicting picture and value clause:** Numeric pictures must have numeric values (no quotes); non-numeric pictures must have non-numeric values (must be enclosed in quotes). Both entries below are *invalid*.

```
05 TOTAL          PIC 9(3)  VALUE "123".
05 TITLE-WORD     PIC X(3)  VALUE 123.
```

Another common error is to use value and picture clauses of different lengths. The entry:

```
05 EMPLOYEE-NAME  PIC X(4)  VALUE "R BAKER".
```

causes a diagnostic for just that reason.

6. **Inadvertent use of COBOL reserved word:** COBOL has a list of some 300 reserved words which can only be used in their designated sense; any other use results in one or several diagnostics. Some reserved words are obvious; e.g., WORKING-STORAGE, IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE. Others such as CODE, DATE, START, and REPORT are less obvious. Instead of memorizing the list or continually referring to it, try this simple rule of thumb. Always use a hyphen in every data name you create. This will work better than 99% of the time.
7. **Conflicting RECORD CONTAINS clause and FD record description:** A recurrent error, even for established programmers. It stems from sometimes careless addition in that the sum of the pictures in an FD does not equal the number of characters in the RECORD CONTAINS clause. It can also result from other errors within the Data Division, namely when an entry containing a PICTURE clause is flagged.
8. **Omitted hyphen in a data name:** A careless error, but one that occurs entirely too often. If in the Data Division we define PRINT-TOTAL-PAY, and then try to reference PRINT TOTAL-PAY, the compiler objects violently. It doesn't state that a hyphen was omitted, but it flags both PRINT and TOTAL-PAY as undefined.

9. *Misspelled data names or reserved words:* Too many COBOL students are poor spellers. Sound strange? How do you spell environment? One or many errors can result, depending on which word was spelled incorrectly.
10. *Reading a record name or writing a file name:* The COBOL rule is very simple. One is supposed to read a file and write a record; many people get it confused. The following entries should clarify the situation:

```
FD EMPLOYEE-FILE
.
.
DATA RECORD IS EMP-RECORD.
.
.
FD PRINT-FILE
.
.
DATA RECORD IS PRINT-RECORD.
```

Correct entries:

```
READ EMPLOYEE-FILE...
WRITE PRINT-RECORD...
```

Incorrect entries:

```
READ EMP-RECORD...
WRITE PRINT-FILE...
```

11. *Going past column 72:* This error can cause any of the preceding errors as well as a host of others to occur. A COBOL statement must end in column 72 or before; columns 73–80 are left blank or used for program identification. If one goes past column 72 in a COBOL statement, it is very difficult to catch because the COBOL listing contains columns 1 to 80 although the compiler only interprets columns 1 to 72.

TRUE/FALSE

1. Data names may contain blanks.
2. If a program compiles correctly, it must execute correctly.
3. In COBOL, one reads a file and writes a record.
4. Compilation stops as soon as one error is found.
5. All compilation error messages appear directly under the statement in error.
6. An error in one statement may cause errors in other, apparently correct statements.
7. It is impossible to execute a program with compilation errors.
8. The compiler will flag all logic errors.
9. Data names in a COMPUTE statement must be defined with numeric pictures.
10. One compile may produce files with extensions of LST and COB.

EXERCISES

1. This problem description is to be used for problems 1 and 2. The Bursar has requested a COBOL program to process a set of student records and calculate the amount due from each student. Incoming records have the following format:

Columns	Field	Picture
1-20	STUDENT NAME	X(20)
21-29	SOCIAL SECURITY NUMBER	9(9)
30-31	CREDITS TAKEN	99
32	UNION MEMBER	X
33-36	SCHOLARSHIP AMOUNT	9(4)

Student bills are calculated as follows:

Tuition:	\$80 per credit.
Union Fee:	\$25 for members, \$50 for nonmembers (members have a "Y" in column 32).
Activity Fee:	\$25 for 6 credits or less. \$50 for 7-12 credits. \$75 for more than 12 credits.
Scholarship:	The amount, if any, is punched in columns 33-36.

The net bill, therefore, is tuition plus union fee plus activity fee, minus scholarship. Correct the compilation errors in Figure 5.5.

TRS-80 Model II COBOL (RM/COBOL 1.3B)
SOURCE FILE: TUICOMP

4/24/81 16.31.13 PAGE 1
OPTION LIST: T L=1

```

LINE  DEBUG PG/LN  A...B.....ID.....
1      000100 IDENTIFICATION DIVISION.
2      000110 PROGRAM-ID.  TUICOMP.
3      000120 AUTHOR.      R GRAUER.
4      000130
5      000140 ENVIRONMENT DIVISION.
6      000150 CONFIGURATION SECTION.
7      000160 SOURCE-COMPUTER.      TRS-80.
8      000170 OBJECT-COMPUTER.    TRS-80.
9      000180
10     000190 INPUT-OUTPUT SECTION.
11     000200 FILE-CONTROL.
12     000210         SELECT STUDENT-FILE
13     000220             ASSIGN TO INPUT "TUITION/DAT".
14     000230         SELECT PRINT-FILE
15     000240             ASSIGN TO PRINT "TUICOMP/TXT".
16     000250
17     000260 DATA DIVISION.
18     000270 FILE SECTION.
19     000280 FD  STUDENT-FILE
20     000290     LABEL RECORDS ARE OMITTED
21     000300     RECORD CONTAINS 80 CHARACTERS
22     000310     DATA RECORD IS STUDENT-RECORD.
23     000320 01  STUDENT-RECORD.
24     000330     05  STUDENT-NAME          PIC X(20).
25     000340     05  SOC-SEC-NUM          PIC 9(9).
26     000350     05  CREDITS              PIC 99.

```

FIGURE 5.5 Debugging exercise

ADDRESS	SIZE	DEBUG	ORDER	TYPE	NAM
	0			FILE	STUDENT-FILE
>0000	80	GRP	0	GROUP	STUDENT-RECORD
>0000	20	ANS	0	ALPHANUMERIC	STUDENT-NAME
>0014	9	NSU	0	NUMERIC UNSIGNED	SOC-SEC-NUM
>001D	2	NSU	0	NUMERIC UNSIGNED	CREDITS
>001F	1	ANS	0	ALPHANUMERIC	UNION-MEMBER
>0020	4	NSU	0	NUMERIC UNSIGNED	SCHOLARSHIP
	0			FILE	PRINT-FILE
>0054	132	GRP	0	GROUP	PRINT-LINE
>0054	20	ANS	0	ALPHANUMERIC	PRINT-STUDENT-NAME
>006A	11	NSE	0	NUMERIC EDITED	PRINT-SOC-SEC-NUM
>0079	2	NSU	0	NUMERIC UNSIGNED	CREDITS
>007E	8	NSE	0	NUMERIC EDITED	PRINT-TUITION
>0087	8	NSE	0	NUMERIC EDITED	PRINT-UNION-FEE
>0092	8	NSE	0	NUMERIC EDITED	PRINT-ACTIVITY-FEE
>009D	8	NSE	0	NUMERIC EDITED	PRINT-SCHOLARSHIP
>00AA	8	NSE	0	NUMERIC EDITED	PRINT-IND-BILL
>00E0	3	ANS	0	ALPHANUMERIC	WS-DATA-REMAINS-SWITCH
>00E4	16	GRP	0	GROUP	IND-COMPUTATIONS
>00E4	4	NSU	0	NUMERIC UNSIGNED	IND-TUITION
>00E8	4	NSU	0	NUMERIC UNSIGNED	IND-ACTIVITY-FEE
>00EC	4	NSU	0	NUMERIC UNSIGNED	IND-UNION-FEE
>00F0	4	NSU	0	NUMERIC UNSIGNED	IND-BILL
>00F4	30	GRP	0	GROUP	UNIVERSITY-TOTALS
>00F4	6	NSU	0	NUMERIC UNSIGNED	TOTAL-TUITION
>00FA	6	NSU	0	NUMERIC UNSIGNED	TOTAL-SCHOLARSHIP
>0100	6	NSU	0	NUMERIC UNSIGNED	TOTAL-ACTIVITY-FEE
>0106	6	NSU	0	NUMERIC UNSIGNED	TOTAL
>010C	6	ANS	0	ALPHANUMERIC	TOTAL-IND-BILL
>0112	132	GRP	0	GROUP	DASHED-LINE
>0196	121	GRP	0	GROUP	HEADING-LINE
READ ONLY BYTE SIZE =				>0228	
READ/WRITE BYTE SIZE =				>028E	
OVERLAY SEGMENT BYTE SIZE =				>0000	
TOTAL BYTE SIZE =				>04B6	
15 ERRORS					
14 WARNINGS					

FIGURE 5.5 Continued

2. The program in Figure 5.6 compiled cleanly, but produced the erroneous report of Figure 5.8. (The desired report is shown in Figure 5.7.) Correct all logic errors so that the program will work as intended.

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID.    TUILOGIC.
000120 AUTHOR.        R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER.   TRS-80.
000170 OBJECT-COMPUTER. TRS-80.

```

FIGURE 5.6 Debugging exercise

```

000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210     SELECT STUDENT-FILE
000220         ASSIGN TO INPUT "TUITION/DAT".
000230     SELECT PRINT-FILE
000240         ASSIGN TO PRINT "TUILOGIC/TXT".
000250
000260 DATA DIVISION.
000270 FILE SECTION.
000280 FD  STUDENT-FILE
000290     LABEL RECORDS ARE OMITTED
000300     RECORD CONTAINS 80 CHARACTERS
000310     DATA RECORD IS STUDENT-RECORD.
000320 01  STUDENT-RECORD.
000330     05  STUDENT-NAME             PIC X(20).
000340     05  SOC-SEC-NUM             PIC 9(9).
000350     05  CREDITS                 PIC 99.
000360     05  UNION-MEMBER           PIC X.
000370     05  SCHOLARSHIP           PIC 9(4).
000380     05  FILLER                 PIC X(44).
000390
000400 FD  PRINT-FILE
000410     LABEL RECORDS ARE STANDARD
000420     RECORD CONTAINS 132 CHARACTERS
000430     DATA RECORD IS PRINT-LINE.
000440 01  PRINT-LINE.
000450     05  PRINT-STUDENT-NAME      PIC X(20).
000460     05  FILLER                 PIC XX.
000470     05  PRINT-SOC-SEC-NUM      PIC 999B99B9999.
000480     05  FILLER                 PIC X(4).
000490     05  PRINT-CREDITS          PIC 99.
000500     05  FILLER                 PIC X(3).
000510     05  PRINT-TUITION          PIC $$$,$$9.
000520     05  FILLER                 PIC X.
000530     05  PRINT-UNION-FEE       PIC $$$,$$9.
000540     05  FILLER                 PIC X(3).
000550     05  PRINT-ACTIVITY-FEE    PIC $$$,$$9.
000560     05  FILLER                 PIC X(3).
000570     05  PRINT-SCHOLARSHIP     PIC      $$9.
000580     05  FILLER                 PIC X(5).
000590     05  PRINT-IND-BILL        PIC $$$,$$9.
000600     05  FILLER                 PIC X(43).
000610
000620 WORKING-STORAGE SECTION.
000630 77  WS-DATA-REMAINS-SWITCH   PIC X(3)      VALUE SPACES.
000640
000650 01  IND-COMPUTATIONS.
000660     05  IND-TUITION            PIC 9(4).
000670     05  IND-ACTIVITY-FEE      PIC 9(4).
000680     05  IND-UNION-FEE        PIC 9(4).
000690     05  IND-BILL             PIC 9(4).
000700
000710 01  UNIVERSITY-TOTALS.
000720     05  TOTAL-TUITION          PIC 9(6)      VALUE ZEROS.
000730     05  TOTAL-SCHOLARSHIP    PIC 9(6)      VALUE ZEROS.
000740     05  TOTAL-ACTIVITY-FEE   PIC 9(6)      VALUE ZEROS.
000750     05  TOTAL-UNION-FEE      PIC 9(6)      VALUE ZEROS.
000760     05  TOTAL-IND-BILL       PIC 9(6)      VALUE ZEROS.
000770
000780 01  DASHED-LINE.
000790     05  FILLER                 PIC X(97)     VALUE ALL "-".
000800     05  FILLER                 PIC X(35)     VALUE SPACES.
000810
000820 01  HEADING-LINE.
000830     05  FILLER                 PIC X(12)     VALUE "STUDENT NAME".
000840     05  FILLER                 PIC X(10)     VALUE SPACES.
000850     05  FILLER                 PIC X(11)     VALUE "SOC SEC NUM".
000860     05  FILLER                 PIC X(2)      VALUE SPACES.
000870     05  FILLER                 PIC X(7)      VALUE "CREDITS".
000880     05  FILLER                 PIC X(2)      VALUE SPACES.
000890     05  FILLER                 PIC X(7)      VALUE "TUITION".
000900     05  FILLER                 PIC X(2)      VALUE SPACES.
000910     05  FILLER                 PIC X(9)      VALUE "UNION FEE".
000920     05  FILLER                 PIC X(2)      VALUE SPACES.
000930     05  FILLER                 PIC X(7)      VALUE "ACT FEE".

```

FIGURE 5.6 Continued

```

000940      05 FILLER                      PIC X(2)      VALUE SPACES.
000950      05 FILLER                      PIC X(11)     VALUE "SCHOLARSHIP".
000960      05 FILLER                      PIC X(2)      VALUE SPACES.
000970      05 FILLER                      PIC X(10)     VALUE "TOTAL BILL".
000980      05 FILLER                      PIC X(36)     VALUE SPACES.
000990
001000  PROCEDURE DIVISION.
001010  PREPARE-TUITION-REPORT.
001020      OPEN INPUT STUDENT-FILE
001030          OUTPUT PRINT-FILE.
001040      PERFORM WRITE-HEADING-LINE.
001070      PERFORM PROCESS-RECORDS
001080          UNTIL WS-DATA-REMAINS-SWITCH = "NO".
001090      PERFORM WRITE-UNIVERSITY-TOTALS.
001100      CLOSE STUDENT-FILE
001110          PRINT-FILE.
001120      STOP RUN.
001130
001140  WRITE-HEADING-LINE.
001150      WRITE PRINT-LINE FROM HEADING-LINE
001160          AFTER ADVANCING PAGE.
001170      WRITE PRINT-LINE FROM DASHED-LINE
001180          AFTER ADVANCING 1 LINE.
001190
001200  PROCESS-RECORDS.
001201      READ STUDENT-FILE
001202          AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
001210      COMPUTE IND-TUITION = 80 * CREDITS.
001220
001230      IF UNION-MEMBER = "Y"
001240          MOVE 25 TO IND-UNION-FEE
001250      ELSE
001260          MOVE ZERO TO IND-UNION-FEE.
001270
001280      MOVE 25 TO IND-ACTIVITY-FEE.
001310      IF CREDITS > 12
001320          MOVE 75 TO IND-ACTIVITY-FEE.
001321      IF CREDITS > 6
001322          MOVE 50 TO IND-ACTIVITY-FEE.
001330
001340      COMPUTE IND-BILL = IND-TUITION + IND-UNION-FEE
001350          + IND-ACTIVITY-FEE - SCHOLARSHIP.
001360
001370      PERFORM UPDATE-UNIVERSITY-TOTALS.
001380      PERFORM WRITE-DETAIL-LINE.
001410
001420  UPDATE-UNIVERSITY-TOTALS.
001430      ADD IND-TUITION TO TOTAL-TUITION.
001450      ADD IND-ACTIVITY-FEE TO TOTAL-ACTIVITY-FEE.
001460      ADD IND-BILL TO TOTAL-IND-BILL.
001470      ADD SCHOLARSHIP TO TOTAL-SCHOLARSHIP.
001480
001490  WRITE-DETAIL-LINE.
001500      MOVE SPACES TO PRINT-LINE.
001510      MOVE STUDENT-NAME TO PRINT-STUDENT-NAME.
001520      MOVE SOC-SEC-NUM TO PRINT-SOC-SEC-NUM.
001530      MOVE CREDITS TO PRINT-CREDITS.
001540      MOVE IND-TUITION TO PRINT-TUITION.
001550      MOVE IND-UNION-FEE TO PRINT-UNION-FEE.
001560      MOVE IND-ACTIVITY-FEE TO PRINT-ACTIVITY-FEE.
001570      MOVE SCHOLARSHIP TO PRINT-SCHOLARSHIP.
001580      MOVE IND-BILL TO PRINT-IND-BILL.
001590      WRITE PRINT-LINE
001600          AFTER ADVANCING 2 LINES.
001610
001620  WRITE-UNIVERSITY-TOTALS.
001630      WRITE PRINT-LINE FROM DASHED-LINE
001640          AFTER ADVANCING 1 LINE.
001650      MOVE SPACES TO PRINT-LINE.
001660      MOVE TOTAL-TUITION TO PRINT-TUITION.
001670      MOVE TOTAL-UNION-FEE TO PRINT-UNION-FEE.
001680      MOVE TOTAL-ACTIVITY-FEE TO PRINT-ACTIVITY-FEE.
001690      MOVE TOTAL-SCHOLARSHIP TO PRINT-SCHOLARSHIP.
001700      MOVE IND-BILL TO PRINT-IND-BILL.
001710      WRITE PRINT-LINE
001720          AFTER ADVANCING 2 LINES.

```

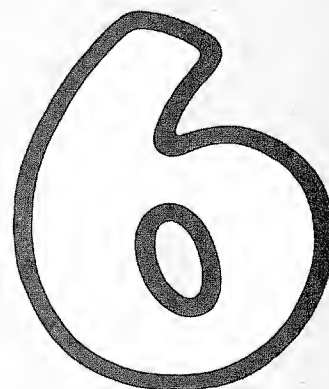
FIGURE 5.6 *Continued*

STUDENT NAME	SOC	SEC	NUM	CREDITS	TUITION	UNION FEE	ACT FEE	SCHOLARSHIP	TOTAL BILL
JOHN SMITH	123	45	6789	15	\$1,200	\$25	\$75	\$0	\$1,300
HENRY JAMES	987	65	4321	15	\$1,200	\$0	\$75	\$500	\$775
SUSAN BAKER	111	22	3333	09	\$720	\$0	\$50	\$500	\$270
JOHN PART-TIMER	456	21	3546	03	\$240	\$25	\$25	\$0	\$290
PEGGY JONES	456	45	6456	15	\$1,200	\$25	\$75	\$0	\$1,300
H. HEAVY-WORKER	789	52	1234	18	\$1,440	\$0	\$75	\$0	\$1,515
BENJAMIN LEE	876	87	6876	18	\$1,440	\$0	\$75	\$0	\$1,515
					\$7,440	\$75	\$450	\$1,000	\$6,965

FIGURE 5.7 Desired report

STUDENT NAME	SOC	SEC	NUM	CREDITS	TUITION	UNION FEE	ACT FEE	SCHOLARSHIP	TOTAL BILL
JOHN SMITH	123	45	6789	15	\$1,200	\$25	\$50	\$0	\$1,275
HENRY JAMES	987	65	4321	15	\$1,200	\$0	\$50	\$0	\$750
SUSAN BAKER	111	22	3333	09	\$720	\$0	\$50	\$0	\$270
JOHN PART-TIMER	456	21	3546	03	\$240	\$25	\$25	\$0	\$290
PEGGY JONES	456	45	6456	15	\$1,200	\$25	\$50	\$0	\$1,275
H. HEAVY-WORKER	789	52	1234	18	\$1,440	\$0	\$50	\$0	\$1,490
BENJAMIN LEE	876	87	6876	18	\$1,440	\$0	\$50	\$0	\$1,490
BENJAMIN LEE	876	87	6876	18	\$1,440	\$0	\$50	\$0	\$1,490
Last record processed twice					\$8,880	\$0	\$375	\$0	\$1,490
					Scholarship should be \$500				Total not correct
					Union fee not summed				

FIGURE 5.8 Invalid tuition report (produced by Figure 5.6)



ADVANCED FEATURES

OVERVIEW The material in Chapters 1 through 5 enables the reader to do some meaningful COBOL programming. The objective of this chapter is to introduce additional COBOL elements which will give the programmer a more powerful grasp of the language. To that end, we cover advanced options of some verbs we already know, as well as present new elements and additional features.

We begin with the IF and PERFORM statements, two of the more powerful verbs. The IF statement is extended to include compound tests, 88-level entries, and nested IFs. Coverage of the PERFORM verb includes the THRU and VARYING options, and differentiates between performing sections versus paragraphs. The ACCEPT statement is used to obtain the date of execution. ACCEPT and DISPLAY are also expanded to include the LINE, POSITION, and REVERSE clauses associated with the CRT.

READ and WRITE are enlarged to cover READ INTO and WRITE FROM, respectively. The ROUNDED and SIZE ERROR options are presented with the COMPUTE verb. Nonunique data names, qualification, and MOVE CORRESPONDING are all included. INSPECT is mentioned as a convenient way of inserting hyphens in a social security number. Finally, there is appreciable coverage of table processing to include the OCCURS and REDEFINES clauses, table initialization, and table lookups.

Needless to say, there is a lot of material in this chapter. We have tried to make it readily understandable by including numerous examples as each feature is introduced. Of greater significance perhaps is the complete program at the chapter's end, which incorporates most of the material and serves as an effective review.

IF The IF statement is one of the more powerful in COBOL. In this section, we expand the discussion to include: compound tests, 88-level entries, and nested IFs. Figures 6.1 and 6.2 illustrate the use of compound conditions:

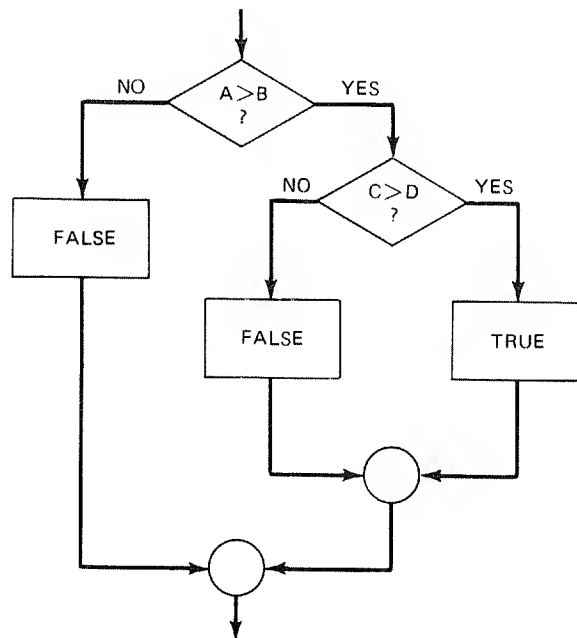


FIGURE 6.1 Flowchart for the condition $A > B$ and $C > D$

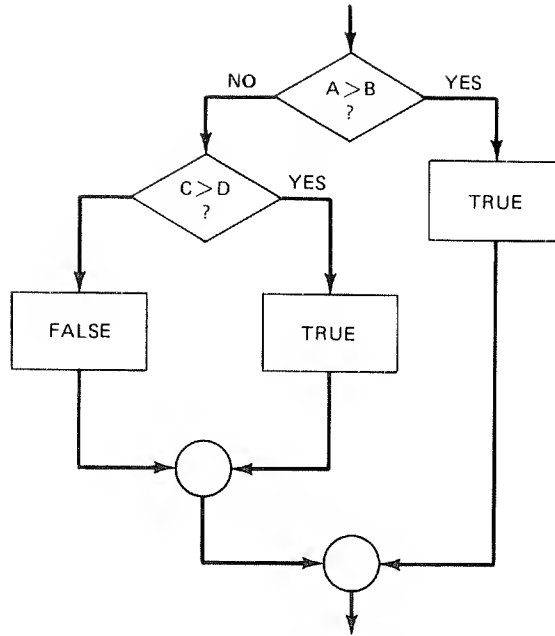


FIGURE 6.2 Flowchart for the condition $A > B$ or $C > D$

Compound Tests

The logical operators, AND and OR may be combined to form a compound test. AND means both; i.e., two conditions must be satisfied for the IF to be considered true. OR means either; i.e., only one of the two conditions need be satisfied for the IF to be considered true. A flowchart is shown in Figure 6.1, depicting the AND condition. It requires that both A be greater than B *and* C be greater than D in order to proceed to TRUE. If either of these tests fails, the compound condition is judged false.

Figure 6.2 contains a flowchart for a compound OR. As can be seen from Figure 6.2, only one of two conditions need be met for the IF to be considered true. If either A is greater than B *or* C is greater than D, processing is directed to TRUE. In other words, the OR provides a second chance in that the first test can fail but the IF can still be considered true.

Beginning programmers are often carried away with compound conditions. Consider the statement:

IF $X > Y$ OR $X = Z$ AND $X < W$...

Surely the programmer knew what was intended at the time the statement was first written. A day later, however, one is apt to stare at it and wonder what will happen first; i.e., which takes precedence, AND or OR? To provide an unequivocal evaluation of compound conditions, COBOL establishes the following hierarchy:

1. Arithmetic expressions
2. Relational operators
3. NOT condition
4. AND (from left to right if more than one)
5. OR (from left to right if more than one)

Thus, for the preceding statement to be true either

$$X > Y$$

or

$$X = Z \text{ and } X < W$$

However, *parentheses can and should be used to clarify the programmer's intent*. The meaning of the above statement is made clearer if it is rewritten as

$$\text{IF } X > Y \text{ OR } (X = Z \text{ AND } X < W) \dots$$

Note well that parentheses can also *alter* meaning. Thus the following statement is logically *different* from the original code:

$$\text{IF } (X > Y \text{ OR } X = Z) \text{ AND } X < W \dots$$

Condition Name Tests (88-Level Entries)

The condition in the IF statement often tests the value of an incoming code, e.g., IF YEAR-IN-SCHOOL = 1 While such coding is quite permissible, and indeed commonplace, the meaning of the value 1 in YEAR-IN-SCHOOL is not immediately apparent. An alternative form of coding, condition names (88-level entries), provides superior documentation. Consider:

05	YEAR-IN-SCHOOL	PIC 9.
88	FRESHMAN	VALUE 1.
88	SOPHOMORE	VALUE 2.
88	JUNIOR	VALUE 3.
88	SENIOR	VALUE 4.
88	GRAD-STUDENT	VALUES ARE 5 THRU 8.
88	UNDER-CLASSMAN	VALUES ARE 1, 2.
88	UPPER-CLASSMAN	VALUES ARE 3, 4.
88	VALID-CODES	VALUES ARE 1 THRU 8.

If the preceding entries were made in the Data Division, one could code

$$\text{IF FRESHMAN}$$

as equivalent to

$$\text{IF YEAR-IN-SCHOOL} = 1$$

The advantage of condition names is threefold. First, they provide improved documentation; i.e., FRESHMAN is inherently clearer than YEAR-IN-SCHOOL = 1. Second, they facilitate maintenance in that additions and/or changes to existing codes need to be made in only one place. For example, suppose the code for freshman is subsequently changed to 10. Only a single change is required in the 88-level entry. If, however, condition names are not used, then one must find all occurrences of YEAR-IN-SCHOOL = 1 in the Procedure Division, and the chance of error is much greater. Finally, they permit *grouping* of several codes into a single entry; e.g., GRAD-STUDENT, and thereby reduce the need for sometimes confusing compound conditions. Thus

IF GRAD-STUDENT ...

is equivalent to

IF YEAR-IN-SCHOOL > 4 AND YEAR-IN-SCHOOL < 9 ...

Nested IFs

The general format of the IF statement is:

$$\text{IF condition} \left\{ \begin{array}{l} \text{statement-1} \\ \text{NEXT SENTENCE} \end{array} \right\} \left[\text{ELSE} \left\{ \begin{array}{l} \text{statement-2} \\ \text{NEXT SENTENCE} \end{array} \right\} \right]$$

The condition may be any of the tests we have discussed, i.e., condition name, relational, or compound. The NEXT SENTENCE clause causes execution to continue with the first statement following the period.

A *nested* IF results when either statement-1 or statement-2 is itself another IF statement; i.e., there are two or more IFs in one sentence. Consider Figure 6.3, which shows a flowchart and corresponding COBOL code to determine the largest of three quantities A, B, and C. (They are assumed to be unequal numbers.)

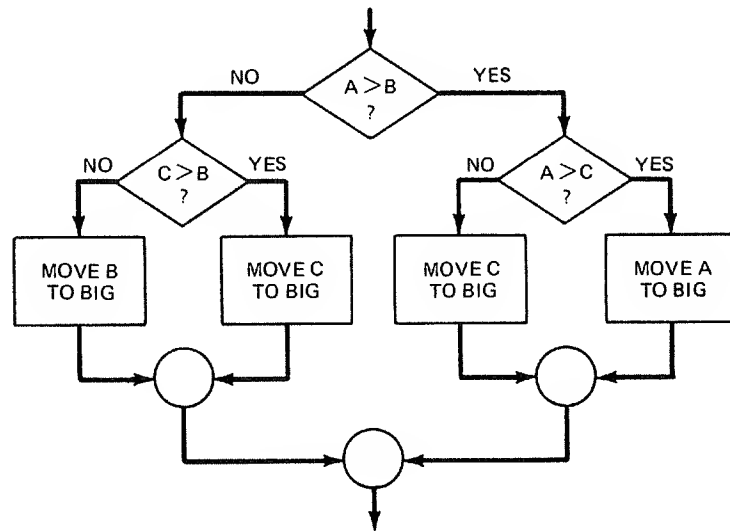


FIGURE 6.3a Flowchart for a nested IF

```

IF A > B
  IF A > C
    MOVE A TO BIG
  ELSE
    MOVE C TO BIG
ELSE
  IF C > B
    MOVE C TO BIG
  ELSE
    MOVE B TO BIG.
  
```

FIGURE 6.3b COBOL code for nested IFs

The code in Figure 6.3b is a nested IF statement because there are three IF clauses within one sentence. The rule for compiler interpretation bears repeating. *The ELSE clause is associated with the closest previous IF that is not already paired with another ELSE.*

The compiler, however, pays no attention to indentation in a nested IF statement, which is done strictly for programmer convenience. We strongly advocate careful attention to indentation and recommend the following guidelines:

1. Each nested IF should be indented four columns from the previous IF.
2. The word ELSE should appear on a line by itself and directly under its associated IF.
3. Detail lines should be indented four columns under both IF and ELSE.

These guidelines were used in Figure 6.3b. A second example follows:

```

IF A > B
  IF C > D
    MOVE S TO W
    MOVE X TO Y
  ELSE
    ADD 1 TO Z.

```

Note that in this example Z is incremented by 1 if A is greater than B, but C is not greater than D. If, however, A is not greater than B, control passes to the next sentence with no further action being taken.

PERFORM The PERFORM verb was introduced in Chapter 4 as the means of implementing a loop. We now consider additional options available with this important verb.

The procedure name in the PERFORM statement can specify either a *paragraph* or a *section*. We already know what a paragraph is. A section consists of one or more paragraphs. If the procedure name in the PERFORM statement refers to a section name, then *every* paragraph in the section will be executed prior to returning control. Consider Figure 6.4.

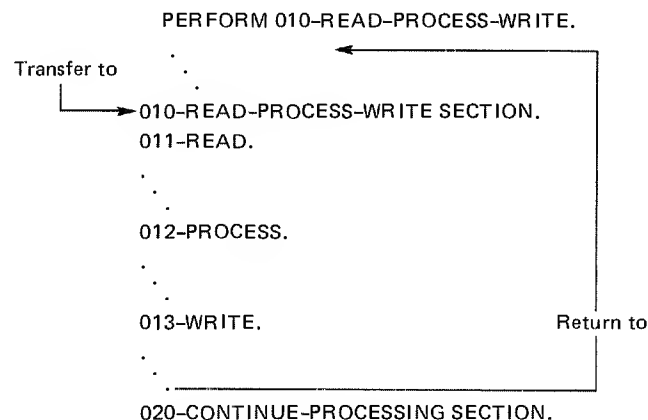


FIGURE 6.4 Performing a section

When the PERFORM statement references a section-name, control is transferred to the first sentence in the section. Control will not return to the sentence after the PERFORM until the last statement in the section was executed. Notice that this results in the execution of several paragraphs. How does the compiler know when the section ends? Simply when a new section name is encountered.

Situations may arise when it is necessary to cease performing, i.e., prematurely exit a given routine. This is accomplished by the THRU option of the PERFORM verb and the EXIT statement. Consider an extended format of the PERFORM:

```
PERFORM procedure-name-1 [THRU procedure-name-2]
```

The THRU option causes *all* statements between the two procedure names to be executed. (Remember, the procedures may be either paragraphs or sections.) Common practice is to make procedure-name-2 a single-sentence paragraph consisting of the word EXIT. The EXIT statement causes no action to be taken; its function is to delineate the end of the PERFORM. Consider Figure 6.5.

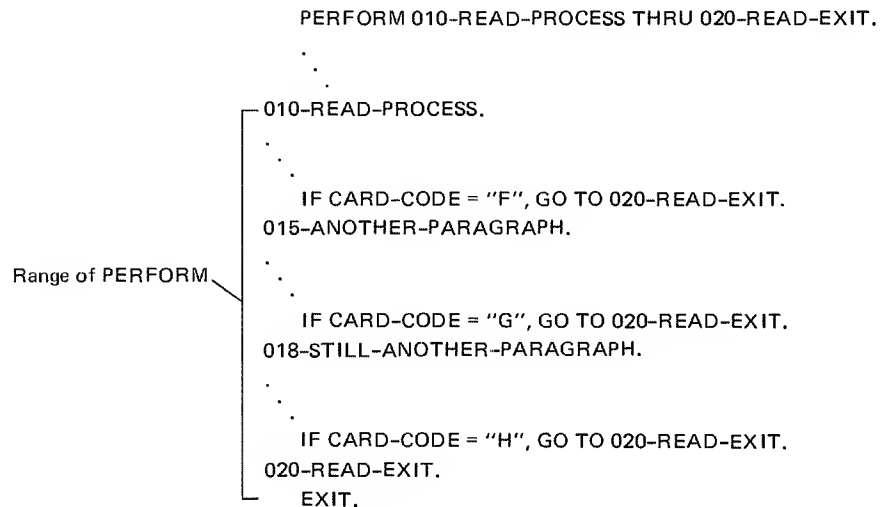


FIGURE 6.5 PERFORM THRU

The PERFORM statement nominally causes execution of all statements within the two procedures. However, it is possible to prematurely terminate the PERFORM by GO TO 020-READ-EXIT. The GO TO statement does not leave the PERFORM, but jumps *forward* to the EXIT statement. (The latter causes no execution, and merely provides a branch point for the GO TO statement.) Hence, the PERFORM is properly terminated, and control returns to the statement after the PERFORM. *Use of a GO TO statement should be severely restricted; i.e., limited to a forward branch to an EXIT statement, as in this example, if not eliminated entirely.* (See Chapter 7 on structured programming.)

Another form of PERFORM, PERFORM/VARYING, will be introduced later in the chapter, in conjunction with table processing.

DISPLAY The DISPLAY statement was first introduced in Chapter 1 as the means of sending information to the CRT terminal. Consider now an abbreviated format:

$$\begin{array}{c} \text{DISPLAY } \left\{ \begin{array}{l} \text{identifier-1} \\ \text{literal-1} \end{array} \right\} \\ \left[\text{LINE } \left\{ \begin{array}{l} \text{identifier-2} \\ \text{literal-2} \end{array} \right\} \right] \\ \left[\text{POSITION } \left\{ \begin{array}{l} \text{identifier-3} \\ \text{literal-3} \end{array} \right\} \right] \\ \left[\text{REVERSE} \right] \dots \end{array}$$

The DISPLAY statement causes either a literal or the value of an identifier to appear on the screen. As can be seen from the three dots in the syntax, more than one literal and/or identifier may appear in the same statement. The LINE option allows the user to specify the particular line (from 1 to 24) where the output is to appear. The POSITION feature specifies the column (from 1 to 80 on a Model II). If LINE is omitted, output will appear on the line immediately below the current position of the cursor. If POSITION is not specified, output is displayed in column 1. REVERSE highlights the result of a DISPLAY statement by printing black letters on a white field as opposed to the normal white letters on a black field. Some examples:

DISPLAY "HELLO".

DISPLAY "X = " X.

DISPLAY "X = ", X, "Y = ", Y.

DISPLAY "GOODBYE" REVERSE.

DISPLAY "TRS-80"
LINE 15 POSITION 20.

The reader is urged to consult the TRS-80 COBOL reference manual for additional options.

ACCEPT The ACCEPT statement is used to obtain the date and/or time of program execution. Consider:

$$\text{ACCEPT identifier-1 FROM } \left\{ \begin{array}{l} \text{DATE} \\ \text{DAY} \\ \text{TIME} \end{array} \right\}$$

In all cases, identifier-1 is a programmer-defined work area to hold the information being accepted. If DATE is specified, then identifier-1 will receive a six-digit numeric field in the form yymmdd. The first two digits contain year, the next two month, and the last two, day of the month, e.g., 790316, denoting March 16, 1979. If DAY, rather than DATE is specified, a five-digit numeric field is returned to the work area. The first two digits represent year and the last three the day of the year, numbered from 1 to 366. March 16, 1979, would be represented as 79075, but March 16, 1980, would be 80076 since 1980 is a leap year.

TIME returns an eight-digit numeric field in a 24-hour system. It contains the number of elapsed hours, minutes, seconds, and hundredths of

seconds after midnight, in that order, from left to right. 10:15 A.M. would return as 10150000; 10:15 P.M. as 22150000.

The importance of *correctly* answering the DATE and TIME questions when powering up the TRS-80 should now become apparent. The ACCEPT statement obtains its values from data initially supplied by the user.

The ACCEPT statement is also the means of inputting data from the keyboard, as was done in the program of Chapter 1. An abbreviated format is:

ACCEPT identifier-1

[LINE {identifier-2}
literal-1}]

[POSITION {identifier-3}
literal-3}]

[REVERSE]...

The LINE, POSITION, and REVERSE options work in a similar fashion as in the DISPLAY statement.

When an ACCEPT statement is executed, the system suspends execution, until a response has been received. Consequently, common practice is to precede any ACCEPT statement with a DISPLAY statement, indicating the nature of the response. (See Figure 1.1.)

READ INTO The general form of the READ statement is:

READ file-name [INTO identifier] AT END imperative statement.

The READ INTO option stores the input record in the specified area and, in addition, moves it to the designated identifier following INTO. Consider:

```
FD EMPLOYEE-FILE
.
.
DATA RECORD IS EMPLOYEE-RECORD.
01 EMPLOYEE-RECORD          PIC X(80).
.
.
WORKING-STORAGE SECTION.
01 WS-RECORD-AREA          PIC X(80).
.
.
PROCEDURE DIVISION.
    READ EMPLOYEE-FILE INTO WS-RECORD-AREA
    AT END PERFORM END-OF-JOB-ROUTINE.
```

The input data will be available in both EMPLOYEE-RECORD and WS-RECORD-AREA. The single READ INTO statement is equivalent to both:

```
READ EMPLOYEE-FILE
    AT END PERFORM END-OF-JOB-ROUTINE.
```

and

```
MOVE EMPLOYEE-RECORD TO WS-RECORD-AREA.
```


WRITE FROM WRITE FROM is analogous to READ INTO in that it combines a MOVE and WRITE statement into one. The general form of the WRITE statement is:

$$\text{WRITE record-name [FROM identifier-1] } \left\{ \begin{array}{l} \text{[BEFORE]} \\ \text{[AFTER]} \end{array} \right\} \text{ADVANCING } \left\{ \begin{array}{l} \text{[identifier-2]} \\ \text{integer} \\ \text{[mnemonic-name]} \\ \text{[PAGE]} \end{array} \right\} \left\{ \begin{array}{l} \text{[LINE]} \\ \text{[LINES]} \end{array} \right\}$$

WRITE FROM is particularly useful when writing heading lines in that VALUE clauses may appear in Working-Storage. Consider:

```
FD PRINT-FILE
.
.
DATA RECORD IS PRINT-LINE.
01 PRINT-LINE                                PIC X(132).
.
.
WORKING-STORAGE SECTION.
01 HEADING-LINE.
   05 FILLER                                PIC X(20)
      VALUE SPACES.
   05 FILLER                                PIC X(12)
      VALUE "ACME WIDGETS".
.
.
WRITE PRINT-LINE FROM HEADING-LINE
AFTER ADVANCING PAGE.
```

ROUNDED AND SIZE ERROR OPTIONS The ROUNDED and SIZE ERROR options are available for the five arithmetic verbs ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE. Both options are frequently used. Consider the general form of the COMPUTE statement:

COMPUTE identifier-1 [ROUNDED] = arithmetic expression
[ON SIZE ERROR imperative-statement]

The SIZE ERROR option signals when the result of a calculation is too large for the designated field. Consider:

```
05 HOURLY-RATE      PIC 99.
05 HOURS-WORKED     PIC 99.
05 GROSS-PAY        PIC 999.
.
.
COMPUTE GROSS-PAY = HOURLY-RATE * HOURS-WORKED.
```

Assume that HOURLY-RATE and HOURS-WORKED are 25 and 40, respectively. The result of the multiplication should be 1000. Unfortunately, because GROSS-PAY was defined as a three-position numeric field, only the three right-most digits are retained, and GROSS-PAY becomes 000. The situation is prevented by the inclusion of the SIZE ERROR option;

COMPUTE GROSS-PAY = HOURLY-RATE * HOURS-WORKED
ON SIZE ERROR PERFORM ERROR-ROUTINE.

Whenever the calculated result is too large, the SIZE ERROR clause will perform ERROR-ROUTINE. The latter consists of programmer-specified logic that should cause a warning message to print. Realize also that SIZE ERROR can be activated by zero division since any attempt to divide by zero results in a quotient of infinity.

The ROUNDED clause causes the last decimal place to be rounded. Consider:

```
COMPUTE GROSS-PAY ROUNDED = HOURLY-RATE * HOURS-WORKED
ON SIZE ERROR PERFORM ERROR-ROUTINE.
```

If GROSS PAY is defined to two decimal places (e.g., PIC 9(4)V99), then .005 is added to the result, and the third decimal place is truncated. Note well that omission of the ROUNDED clause results in the last decimal being truncated, rather than rounded.

DUPLICATE DATA NAMES

Most programs require that the output contain some of the input, e.g., name, social security number, etc. COBOL permits *duplicate* data names to be defined in the Data Division provided all Procedure Division references to duplicate data names use *qualification*. Duplicate names are conducive to the CORRESPONDING option, which results in fewer statements in the Procedure Division. Both qualification and the CORRESPONDING option are discussed in accordance with Figure 6.6.

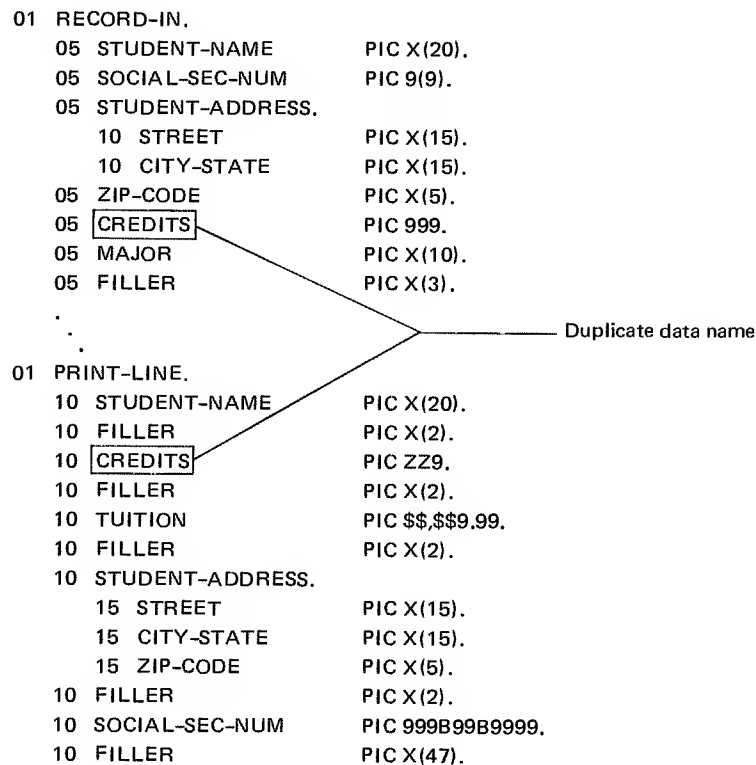


FIGURE 6.6 Data Division code for duplicate data names

Qualification

The coding in Figure 6.6 has several duplicate data names in RECORD-IN and PRINT-LINE, e.g., CREDITS, and it is confusing to reference any of these data names in the Procedure Division. Consider the statement:

MULTIPLY CREDITS BY COST-PER-CREDIT GIVING CHARGE.

The use of CREDITS is ambiguous; i.e., the compiler does not know which CREDITS (i.e., in RECORD-IN or PRINT-LINE) we are talking about. The solution is to qualify the data name, using OF or IN to clarify the reference. Thus the statement is rewritten as:

MULTIPLY CREDITS OF RECORD-IN BY COST-PER-CREDIT GIVING CHARGE.

Qualification may be required over several levels. For example, this statement is still ambiguous:

MOVE STREET OF STUDENT-ADDRESS TO OUTPUT-AREA.

Both STREET and STUDENT-ADDRESS are duplicate data names, so the qualification didn't help. We could use two levels to make our intent clear, e.g.,

MOVE STREET OF STUDENT-ADDRESS OF RECORD-IN TO OUTPUT-AREA.

We could also skip the intermediate level and code:

MOVE STREET IN RECORD-IN TO OUTPUT-AREA.

Notice that OF and IN can be used interchangeably. Duplicate data names offer the advantage of not having to invent different names for the same item, e.g., an employee name appearing in both an input record and output record. They also permit use of the CORRESPONDING option.

CORRESPONDING

The general form of the CORRESPONDING option is

$$\text{MOVE } \left\{ \begin{array}{c} \text{CORRESPONDING} \\ \text{CORR} \end{array} \right\} \text{ identifier-1 TO identifier-2}$$

Notice that CORR is the abbreviated form of CORRESPONDING (analogous to PIC and PICTURE). Consider the record description in Figure 6.6 and the statement:

MOVE CORRESPONDING RECORD-IN TO PRINT-LINE.

The MOVE CORRESPONDING statement is equivalent to several individual MOVES. It takes every data name of RECORD-IN and looks for a duplicate data name in PRINT-LINE. Whenever a "match" is found, an individual MOVE is generated. Thus the preceding MOVE CORRESPONDING is equivalent to:

MOVE STUDENT-NAME OF RECORD-IN	TO STUDENT-NAME OF PRINT-LINE.
MOVE SOCIAL-SEC-NUM OF RECORD-IN	TO SOCIAL-SEC-NUM OF PRINT-LINE.
MOVE STREET OF RECORD-IN	TO STREET OF PRINT-LINE.
MOVE CITY-STATE OF RECORD-IN	TO CITY-STATE OF PRINT-LINE.
MOVE CREDITS OF RECORD-IN	TO CREDITS OF PRINT-LINE.

Notice that the level numbers of the duplicate data names do *not* have to match; it is only the data names themselves that must be the same in each record. Further, notice that the order of the data names is immaterial; e.g., SOCIAL-SEC-NUM is the second field in RECORD-IN and the next to last in PRINT-LINE.

Care must be taken to observe the following restrictions pertaining to the CORRESPONDING option. In particular:

1. At least one item in each pair of CORRESPONDING items must be an elementary item for the MOVE to be effective. Thus, in the example, STUDENT-ADDRESS of RECORD-IN is not moved to STUDENT-ADDRESS of PRINT-LINE. (The elementary items STREET and CITY-STATE are moved instead.)
2. Corresponding elementary items will be moved only if they have the same name and qualification up to but not including identifier-1 and identifier-2. Thus ZIP-CODE will *not* be moved.

INSPECT The INSPECT statement is often used to insert hyphens in a social security number. Assume, for example, that social security number is stored as a nine-position field (with no hyphens) in an input record, but that it is to appear with hyphens in a printed report. This is accomplished as follows:

```

01 RECORD-IN.
.
.
05 SOC-SEC-NUM          PIC 9(9).
01 PRINT-LINE.
.
.
05 SOC-SEC-NUM-OUT      PIC 999B99B9999.

PROCEDURE DIVISION.
.
.
MOVE SOC-SEC-NUM TO SOC-SEC-NUM-OUT.
INSPECT SOC-SEC-NUM-OUT REPLACING ALL " " BY "-".

```

The MOVE statement transfers the incoming social security number to an 11-position field containing two blanks (denoted by B in the PICTURE clause). The INSPECT statement replaces every occurrence of a blank in SOC-SEC-NUM-OUT by the desired hyphen. (This technique is also used to insert /'s in date fields.)

TABLES A table is a grouping of similar data. The values in a table are stored in consecutive storage locations and assigned a single data name. Reference to individual items within a table is accomplished by subscripts that identify the location of the particular item.

For example, assume company XYZ tabulates its sales on a monthly basis and that the sales of each month have to be referenced within a COBOL program. Without tables, 12 data names are required: SALES-FOR-JANUARY, SALES-FOR-FEBRUARY, etc. With tables, however, we define only a single data name, e.g., SALES, and refer to individual months by an appropriate subscript. Thus SALES (2) would indicate sales for the second month, i.e., February.

Figure 6.7 is an example of a table with 12 entries.

<i>Month</i>	<i>Sales</i>
Jan.	\$1000
Feb.	\$2000
Mar.	\$3000
April	\$4000
May	\$5000
June	\$4000
July	\$3000
Aug.	\$2000
Sept.	\$1000
Oct.	\$2000
Nov.	\$3000
Dec.	\$6000

Note: SALES (3) = sales
for 3rd month = \$3000.

FIGURE 6.7 One-level table

The OCCURS Clause

The OCCURS clause specifies the number of entries in a table. The format of the OCCURS clause is simply:

OCCURS integer TIMES

Thus, for the one-dimension table of Figure 6.7, we might have the entry:

```
05 SALES OCCURS 12 TIMES PIC 9(6).
```

This entry would cause a 72-position table (12 entries \times 6 positions per entry) to be established in the computer's memory, as shown:

SALES																									
SALES (1)						SALES (2)												SALES (12)							

There may be instances in which the OCCURS clause functions as a group item and does not contain a PICTURE clause. Consider:

```
05 SALES-TABLE OCCURS 12 TIMES.
10 VOLUME PIC 9(6).
10 MONTH PIC X(10).
```

SALES-TABLE contains 192 (12 \times 16) positions, and is shown schematically:

The value of SUBSCRIPT is initialized to 1 and automatically incremented by 1 every time the paragraph COMPUTE-ANNUAL-TOTALS is executed. Look carefully at the VARYING and UNTIL clauses; VARYING SUBSCRIPT FROM 1 BY 1 UNTIL SUBSCRIPT > 12. The greater-than sign causes the paragraph COMPUTE-ANNUAL-TOTALS to be executed 12 times. (PERFORM VARYING *increments, tests, and then branches*. Accordingly, if an equal sign had been used instead; i.e., UNTIL SUBSCRIPT = 12, the paragraph would have been executed only 11 times.)

COBOL subscripts may be either variable or constant, and must adhere to the following rules:

1. A space may not precede the right parenthesis nor follow the left parenthesis.

```
VALID:    SALES (SUB)
VALID:    SALES (2)
INVALID:  SALES ( 2)
INVALID:  SALES (2 )
```

2. At least one space is required between the data name and left parenthesis.

```
INVALID:  SALES(SUB)
VALID:    SALES (2)
INVALID:  SALES(2)
```

REDEFINES Clause

The REDEFINES clause is frequently used to establish constant values for a table. Assume, for example, that it is necessary to refer to the 12 months of the year by name. A person knows that MONTH (1) refers to January, MONTH (2) to February, etc. but the computer must be made aware of this explicitly. The OCCURS and REDEFINES clauses are used in conjunction with one another in Working-Storage as shown in Figure 6.8.

The group item MONTH-NAMES has 12 FILLER entries, each 10 characters long. The VALUE clause is used with each FILLER entry to establish an initial value.

WORKING-STORAGE SECTION.

```

01 MONTH-TABLE.
   05 MONTH-NAMES.
      10 FILLER          PIC X(10) VALUE "JANUARY ".
      10 FILLER          PIC X(10) VALUE "FEBRUARY ".
      10 FILLER          PIC X(10) VALUE "MARCH   ".
      10 FILLER          PIC X(10) VALUE "APRIL   ".
      10 FILLER          PIC X(10) VALUE "MAY     ".
      10 FILLER          PIC X(10) VALUE "JUNE    ".
      10 FILLER          PIC X(10) VALUE "JULY    ".
      10 FILLER          PIC X(10) VALUE "AUGUST   ".
      10 FILLER          PIC X(10) VALUE "SEPTEMBER".
      10 FILLER          PIC X(10) VALUE "OCTOBER ".
      10 FILLER          PIC X(10) VALUE "NOVEMBER".
      10 FILLER          PIC X(10) VALUE "DECEMBER ".
   05 MONTH-SUB REDEFINES MONTH-NAMES.
      10 MONTH OCCURS 12 TIMES PIC X(10).
```

FIGURE 6.8 Initialization of a table using OCCURS and REDEFINES

The REDEFINES clause gives another name to previously allocated space; i.e., MONTH-SUB is another name for the 120 positions of MONTH-NAMES. However, MONTH-SUB consists of a table, MONTH, with 12 entries. Thus, MONTH(1) refers to the first 10 positions in MONTH-NAMES ("JANUARY "), MONTH(2) to the next 10 positions ("FEBRUARY "), etc. This may appear somewhat confusing, but it is made mandatory by a language restriction in COBOL: a *VALUE* clause cannot be used in the same statement as an *OCCURS* clause.

Table Lookups

Data are almost invariably stored in coded rather than expanded format. The obvious advantage is that less space is required in the storage medium, e.g., the diskette containing the data file. Thus, somewhere in the program, a conversion from a code to an expanded value has to take place. The conversion is known as a *table lookup* and is illustrated in Figure 6.9.

WORKING-STORAGE SECTION.

```
01 TABLE-PROCESSING-ELEMENTS.
    05 WS-MAJOR-SUB          PIC 9(4).
    05 WS-FOUND-MAJOR-SWITCH PIC X(3).
01 MAJOR-VALUE.
    05 FILLER                PIC X(14) VALUE "1234ACCOUNTING ".
    05 FILLER                PIC X(14) VALUE "1400BIOLOGY   ".
    05 FILLER                PIC X(14) VALUE "1978CHEMISTRY  ".
    05 FILLER                PIC X(14) VALUE "2100CIVIL ENG   ".
    05 FILLER                PIC X(14) VALUE "2458E. D. P.    ".
    05 FILLER                PIC X(14) VALUE "3245ECONOMICS  ".
    05 FILLER                PIC X(14) VALUE "3960FINANCE    ".
    05 FILLER                PIC X(14) VALUE "4321MANAGEMENT".
    05 FILLER                PIC X(14) VALUE "4999MARKETING  ".
    05 FILLER                PIC X(14) VALUE "5400STATISTICS ".
01 MAJOR-TABLE REDEFINES MAJOR-VALUE.
    05 MAJORS OCCURS 10 TIMES.
        10 MAJOR-CODE      PIC X(4).
        10 MAJOR-NAME      PIC X(10).
```

PROCEDURE DIVISION.

```
    MOVE 1 TO WS-MAJOR-SUB.
    MOVE "NO" TO WS-FOUND-MAJOR-SWITCH.
    PERFORM 030-FIND-MAJOR
        UNTIL WS-FOUND-MAJOR-SWITCH = "YES".
```

030-FIND-MAJOR.

```
    IF WS-MAJOR-SUB > 10
        MOVE "YES" TO WS-FOUND-MAJOR-SWITCH
        MOVE "UNKNOWN" TO HDG-MAJOR
    ELSE
        IF ST-MAJOR-CODE = MAJOR-CODE (WS-MAJOR-SUB)
            MOVE "YES" TO WS-FOUND-MAJOR-SWITCH
            MOVE MAJOR-NAME (WS-MAJOR-SUB) TO HDG-MAJOR
        ELSE
            ADD 1 TO WS-MAJOR-SUB.
```

FIGURE 6.9 Table lookup

The objective of Figure 6.9 is to take a four-digit code for major and convert it to an expanded value that will subsequently appear in a printed report. The table of codes and expanded values is established in Working-Storage using the OCCURS, REDEFINES, and VALUE clauses discussed earlier.

The logic in Figure 6.9 begins by setting a subscript to 1 and a switch to 'NO'. When the incoming code, ST-MAJOR-CODE, matches a code in the table, the switch, WS-FOUND-MAJOR-SWITCH, is set to 'YES' and processing is finished. Notice that the value of the subscript, WS-MAJOR-SUB, is compared to the number of entries in the table. If the entire table has been checked without finding a match, we signify an unknown major and terminate processing in the loop. This type of error checking is extremely important and is one way of distinguishing the professional from the student. If the check were not included and an unknown code did appear, the subscript would be incremented indefinitely until some type of fatal error occurred.

SUMMARY This chapter covered several advanced Procedure Division capabilities. We began with the IF statement and several of its ramifications. We studied the INSPECT verb and saw its use in editing data. We took another look at the I/O statements ACCEPT, DISPLAY, READ INTO, and WRITE FROM. We learned about nonunique data names, qualification, and the CORRESPONDING option. Arithmetic statements were expanded to include the ROUNDED and SIZE ERROR options. We took a second look at the PERFORM verb, learned how to define a table and to code a "table lookup" procedure.

While we hope the material in this chapter has been understandable, we readily admit it can make for dry reading. Our fundamental approach throughout the book is to learn by doing. To that end we have developed a complete COBOL program that incorporates most of the material in this unit. Specifications are as follows:

Input	A file of customer records in a car rental agency containing the fields: customer name, car type (i.e., compact, intermediate, or full size), car make (e.g., Buick, Datsun, etc.), days rented, and miles driven.
Processing	Compute the amount of money owed by each customer. The amount is a function of car type, days rented, and miles driven. Compact cars are billed at 12¢ a mile and \$11.00 a day, intermediate cars at 14¢ a mile and \$12.00 a day, and full size cars at 16¢ a mile and \$13.00 a day. In addition, each incoming record is to be checked for valid car type (C, I, or F) and valid car make (according to a table in the program). Records with invalid data are to be flagged and not processed further.
Output	One line of information for each valid record. Each detail record is to include the car make,

expanded from the input code, necessitating a table lookup routine. Output is to be double spaced and limited to five customers on a page, which requires establishment of a suitable page heading routine.

The completed COBOL program is shown in Figure 6.10. The record layouts for CAR-RENTAL-FILE and PRINT-FILE are specified in Working-Storage rather than the File Section. Accordingly, note the use of subsequent READ INTO and WRITE FROM statements (lines 1530 and 2060).

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID. CARS.
000120 AUTHOR. R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER. TRS-80.
000170 OBJECT-COMPUTER. TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210     SELECT CAR-RENTAL-FILE
000220     ASSIGN TO INPUT "CARS/DAT".
000230     SELECT PRINT-FILE
000240     ASSIGN TO PRINT "CARS/TXT".
000250
000260 DATA DIVISION.
000270 FILE SECTION.
000280 FD CAR-RENTAL-FILE
000290     LABEL RECORDS ARE OMITTED
000300     RECORD CONTAINS 50 CHARACTERS
000310     DATA RECORD IS CAR-RENTAL-RECORD.
000320 01 CAR-RENTAL-RECORD          PIC X(50).
000330
000340 FD PRINT-FILE
000350     LABEL RECORDS ARE STANDARD
000360     RECORD CONTAINS 132 CHARACTERS
000370     DATA RECORD IS PRINT-LINE.
000380 01 PRINT-LINE                 PIC X(132).
000390
000400 WORKING-STORAGE SECTION.
000410 01 PROGRAM-SWITCHES.
000420     05 WS-DATA-REMAINS-SWITCH PIC X(2)      VALUE SPACES.
000430     88 NO-MORE-RECORDS        VALUE "NO".
000440     05 WS-CAR-MAKE-SWITCH     PIC X(3)      VALUE SPACES.
000450     88 CAR-CODE-EXPANDED      VALUE "YES".
000460
000470 01 PROGRAM-SUBSCRIPTS.
000480     05 CAR-MAKE-SUBSCRIPT      PIC 99.
000490
000500 01 DATE-WORK-AREA.
000510     05 TODAYS-YEAR            PIC 99.
000520     05 TODAYS-MONTH          PIC 99.
000530     05 TODAYS-DAY            PIC 99.
000540
000550 01 WS-RECORD-IN.
000560     05 SOC-SEC-NUM            PIC 9(9).
000570     05 CUST-NAME              PIC X(25).
000580     05 DATE-RETURNED          PIC 9(6).
000590     05 CAR-TYPE              PIC X.
000600     88 COMPACT                VALUE "C".
000610     88 INTERMEDIATE           VALUE "I".
000620     88 FULL-SIZE              VALUE "F".

```

Definition of 88 level entries

FIGURE 6.10 Car billing problem

```

000630      88  VALID-TYPE-CODE                VALUES "C" "I" "F".
000640      05  CAR-MAKE                      PIC X(3).
000650      88  VALID-MAKE-CODE
000660          VALUES ARE "BUI" "CAD" "CHE" "CHR" "DAT"
000670          "FOR" "OLD" "PON" "TOY" "VOL".
000680      05  DAYS-RENTED                    PIC 99.
000690      05  MILES-DRIVEN                  PIC 9(4).
000700
000710  01  WS-PRINT-LINE.
000720      05  FILLER                        PIC X(4).
000730      05  SOC-SEC-NUM                  PIC 999B999B9999.
000740      05  FILLER                        PIC X(4).
000750      05  CUST-NAME                    PIC X(25).
000760      05  FILLER                        PIC X(2).
000770      05  CAR-TYPE                     PIC X.
000780      05  FILLER                        PIC X(4).
000790      05  PRINT-CAR-MAKE               PIC X(10).
000800      05  FILLER                        PIC XX.
000810      05  DAYS-RENTED                    PIC Z9.
000820      05  FILLER                        PIC X(4).
000830      05  MILES-DRIVEN                  PIC ZZZ9.
000840      05  FILLER                        PIC X(4).
000850      05  CUSTOMER-BILL                 PIC $$,$$9.99.
000860      05  FILLER                        PIC X(46).
000861
000870  01  CAR-MAKE-VALUES.
000880      05  FILLER
000890      05  FILLER
000900      05  FILLER
000910      05  FILLER
000920      05  FILLER
000930      05  FILLER
000940      05  FILLER
000950      05  FILLER
000960      05  FILLER
000970      05  FILLER
000980
000990  01  CAR-MAKE-TABLE REDEFINES CAR-MAKE-VALUES.
001000      05  CAR-CODE-AND-VALUE OCCURS 10 TIMES.
001010          10  CAR-CODE                  PIC X(3).
001020          10  EXPANDED-CAR-MAKE          PIC X(10).
001030
001040  01  PAGE-AND-LINE-COUNTERS.
001050      05  WS-PAGE-COUNT                    PIC 9(4)    VALUE ZEROS.
001060      05  WS-LINE-COUNT                    PIC 9(4)    VALUE 6.
001070
001080  01  BILLING-CONSTANTS.
001090      05  WS-MILEAGE-RATE                    PIC 9V99.
001100      05  WS-DAILY-RATE                    PIC 99V99.
001110      05  WS-CUSTOMER-BILL                 PIC 9(4)V99.
001120
001130  01  HEADING-LINE-ONE.
001140      05  FILLER                        PIC X(75)    VALUE SPACES.
001150      05  FILLER                        PIC X(4)     VALUE "PAGE".
001160      05  WS-PAGE-PRINT                    PIC Z(4).
001170      05  FILLER                        PIC X(49)    VALUE SPACES.
001180
001190  01  HEADING-LINE-TWO.
001200      05  FILLER                        PIC X(25)    VALUE SPACES.
001210      05  TITLE-INFORMATION              PIC X(27)
001220          VALUE "JESSICA'S CAR RENTAL AGENCY".
001230      05  FILLER                        PIC X(2)     VALUE SPACES.
001240      05  TITLE-DATE.
001250          10  TITLE-MONTH                  PIC 99.
001260          10  FILLER                        PIC X.
001270          10  TITLE-DAY                    PIC 99.
001280          10  FILLER                        PIC X.
001290          10  TITLE-YEAR                    PIC 99.
001300      05  FILLER                        PIC X(70)    VALUE SPACES.
001310
001320  01  HEADING-LINE-THREE.

```

Use of blank as editing character

Table initialization

PIC X(13)	VALUE	"BUIBUICK	"
PIC X(13)	VALUE	"CADCADALAC	"
PIC X(13)	VALUE	"CHECHEVROLET	"
PIC X(13)	VALUE	"CHRCRYSLER	"
PIC X(13)	VALUE	"DATDATSUN	"
PIC X(13)	VALUE	"FORFORD	"
PIC X(13)	VALUE	"OLDOLDSMOBILE"	"
PIC X(13)	VALUE	"PONPONTIAC	"
PIC X(13)	VALUE	"TOYTOYOTA	"
PIC X(13)	VALUE	"VOLVOLKSWAGON"	"

Definition of a table

Definition of multiple heading lines

FIGURE 6.10 Continued

001330	05	FILLER	PIC X(5)	VALUE SPACES.	
001340	05	FILLER	PIC X(11)	VALUE " ACCT #".	
001350	05	FILLER	PIC X(5)	VALUE SPACES.	
001360	05	FILLER	PIC X(4)	VALUE "NAME".	
001370	05	FILLER	PIC X(19)	VALUE SPACES.	
001380	05	FILLER	PIC X(4)	VALUE "TYPE".	
001390	05	FILLER	PIC X(4)	VALUE SPACES.	
001400	05	FILLER	PIC X(4)	VALUE "MAKE".	
001410	05	FILLER	PIC X(6)	VALUE SPACES.	
001420	05	FILLER	PIC X(4)	VALUE "DAYS".	
001430	05	FILLER	PIC X(2)	VALUE SPACES.	
001440	05	FILLER	PIC X(5)	VALUE "MILES".	
001445	05	FILLER	PIC X(8)	VALUE SPACES.	
001450	05	FILLER	PIC X(6)	VALUE "AMOUNT".	
001460	05	FILLER	PIC X(45)	VALUE SPACES.	
001470					
001480		PROCEDURE DIVISION.			
001490	100	PREPARE-CAR-BILLING-REPORT.			Obtains date of execution
001500		ACCEPT DATE-WORK-AREA FROM DATE.			
001510		OPEN INPUT CAR-RENTAL-FILE			
001520		OUTPUT PRINT-FILE.			
001530		READ CAR-RENTAL-FILE INTO WS-RECORD-IN			
001540		AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.			
001550		PERFORM 200-PROCESS-RECORDS			
001560		UNTIL NO-MORE-RECORDS.			
001570		CLOSE CAR-RENTAL-FILE			
001580		PRINT-FILE.			
001590		STOP RUN.			
001600					
001610	200	PROCESS-RECORDS.			Compound test with 88 level entries
001620		IF VALID-TYPE-CODE AND VALID-MAKE-CODE			
001630		PERFORM 300-COMPUTE-CAR-BILL			
001640		PERFORM 400-WRITE-DETAIL-LINE			
001650		ELSE			Use of qualification
001660		DISPLAY "ERROR IN INCOMING CAR TYPE OR CAR MAKE "			
001670		CUST-NAME OF WS-RECORD-IN.			
001680					
001690		READ CAR-RENTAL-FILE INTO WS-RECORD-IN			
001700		AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.			
001710					
001720	300	COMPUTE-CAR-BILL.			
001730		IF COMPACT			Nested IF determines appropriate rate
001740		MOVE .12 TO WS-MILEAGE-RATE			
001750		MOVE 11.00 TO WS-DAILY-RATE			
001760		ELSE			
001770		IF INTERMEDIATE			
001780		MOVE .14 TO WS-MILEAGE-RATE			
001790		MOVE 12.00 TO WS-DAILY-RATE			
001800		ELSE			
001810		MOVE .16 TO WS-MILEAGE-RATE			
001820		MOVE 13.00 TO WS-DAILY-RATE.			
001830					
001840		COMPUTE WS-CUSTOMER-BILL <u>ROUNDED</u> =			
001850		MILES-DRIVEN OF WS-RECORD-IN * WS-MILEAGE-RATE			
001860		+ DAYS-RENTED OF WS-RECORD-IN * WS-DAILY-RATE			
001870		<u>ON SIZE ERROR</u>			
001880		DISPLAY "RECEIVING FIELD TOO SMALL FOR AMOUNT DUE"			
001890		CUST-NAME OF WS-RECORD-IN.			
001900					
001910	400	WRITE-DETAIL-LINE.			ROUNDED and SIZE ERROR options
001920		IF WS-LINE-COUNT > 5			
001930		PERFORM 600-WRITE-HEADING.			
001940		MOVE SPACES TO WS-PRINT-LINE.			INSPECT statement inserts hyphens in social security number
001950		MOVE CORRESPONDING WS-RECORD-IN TO WS-PRINT-LINE.			
001960		INSPECT SOC-SEC-NUM OF WS-PRINT-LINE			
001970		REPLACING ALL " " BY "-".			
001980		MOVE WS-CUSTOMER-BILL TO CUSTOMER-BILL.			
001990					
002000		MOVE "NO" TO WS-CAR-MAKE-SWITCH.			Use of PERFORM VARYING
002010		PERFORM 500-EXPAND-CAR-MAKE-CODE			
002020		VARYING CAR-MAKE-SUBSCRIPT FROM 1 BY 1			

FIGURE 6.10 Continued

```

002030          UNTIL CAR-MAKE-SUBSCRIPT > 10
002040          OR CAR-CODE-EXPANDED.
002050
002060      WRITE PRINT-LINE FROM WS-PRINT-LINE
002070      AFTER ADVANCING 2 LINES.
002080      ADD 1 TO WS-LINE-COUNT.
002090
002100 500-EXPAND-CAR-MAKE-CODE. Table lookup is controlled by PERFORM VARYING
002110      IF CAR-MAKE = CAR-CODE (CAR-MAKE-SUBSCRIPT)
002120      MOVE EXPANDED-CAR-MAKE (CAR-MAKE-SUBSCRIPT)
002130      TO PRINT-CAR-MAKE
002140      MOVE "YES" TO WS-CAR-MAKE-SWITCH.
002150
002160 600-WRITE-HEADING.
002170      ADD 1 TO WS-PAGE-COUNT.
002180      MOVE 1 TO WS-LINE-COUNT. Resets line count
002190      MOVE WS-PAGE-COUNT TO WS-PAGE-PRINT.
002200      WRITE PRINT-LINE FROM HEADING-LINE-ONE
002210      AFTER ADVANCING PAGE.
002220      MOVE TODAYS-DAY TO TITLE-DAY.
002230      MOVE TODAYS-MONTH TO TITLE-MONTH.
002240      MOVE TODAYS-YEAR TO TITLE-YEAR.
002250      WRITE PRINT-LINE FROM HEADING-LINE-TWO
002260      AFTER ADVANCING 2 LINES.
002270      WRITE PRINT-LINE FROM HEADING-LINE-THREE
002280      AFTER ADVANCING 2 LINES.
002290

```

FIGURE 6.10 Continued

The definition of WS-RECORD-IN includes 88-level entries for both CAR-TYPE and CAR-MAKE, with several codes grouped under the same condition name. This greatly simplifies the IF statement of line 1620, which checks for valid codes.

A table of car makes is established in lines 870 through 1020. Values are assigned to successive locations in the 01 entry CAR-MAKE-VALUES, after which the table itself is established through the OCCURS and REDEFINES clauses. The table lookup routine is controlled by the PERFORM VARYING statement of lines 2010-2040. The paragraph 500-EXPAND-CAR-MAKE-CODE is entered a maximum of 10 times. It checks if the incoming code matches the current entry in a table. If a match is found, the corresponding expanded value is moved to a print line and the lookup is terminated. Note well how WS-CAR-MAKE-SWITCH is set to "NO" immediately before the PERFORM statement in line 2000.

A nested IF statement (lines 1730-1820) determines appropriate values for WS-MILEAGE-RATE and WS-DAILY-RATE, which are referenced in the COMPUTE of lines 1840-1890. Observe the use of both ROUNDED and SIZE ERROR clauses.

WS-LINE-COUNT is established in Working-Storage (line 1060) to keep control of the number of lines per page. This value is checked prior to writing a detail line in lines 1920-1930. If it exceeds 5 (because 5 lines are required per page), then the paragraph 600-WRITE-HEADING is invoked. This in turn resets WS-LINE-COUNT to one, as well as writes a three-line heading on a new page. Note well that HEADING-LINE-TWO contains the date of execution, which was obtained from the ACCEPT statement of line 1500.

Test data are shown in Figure 6.11 and corresponding output in Figure 6.12. Error messages for the invalid records of Figure 6.11 were displayed on the CRT.

111111111ADAMS, JOHN	020181CTOY051000	
222222222BOROW, JEFF	020981CDAT100986	
333333333LEE, BENJAMIN	013181ICHE010035	
444444444MILGROM, MARION	020881FCHE010042	
555555555GRAUER, SAM	020481FFOR020345	
666666666SUGRUE, PAUL	020981FPLY040601	Invalid car make Invalid car type
777777777CRAWFORD, MARSHAL	020881XCAD030588	
888888888GOODMAN, NEIL	020181CDAT010014	
999999999GULFMAN, STEVEN	020681IPON060510	

FIGURE 6.11 Test data for car billing problem

JESSICA'S CAR RENTAL AGENCY 04/24/81							PAGE 2
ACCT #	NAME	TYPE	MAKE	DAYS	MILES	AMOUNT	
888-88-8888	GOODMAN, NEIL	C	DATSUN	1	14	\$12.68	
999-99-9999	GULFMAN, STEVEN	I	PONTIAC	6	510	\$143.40	

JESSICA'S CAR RENTAL AGENCY 04/24/81							PAGE 1
ACCT #	NAME	TYPE	MAKE	DAYS	MILES	AMOUNT	
111-11-1111	ADAMS, JOHN	C	TOYOTA	5	1000	\$175.00	
222-22-2222	BOROW, JEFF	C	DATSUN	10	986	\$228.32	
333-33-3333	LEE, BENJAMIN	I	CHEVROLET	1	35	\$16.90	
444-44-4444	MILGROM, MARION	F	CHEVROLET	1	42	\$19.72	
555-55-5555	GRAUER, SAM	F	FORD	2	345	\$81.20	

FIGURE 6.12 Output of car billing problem. (Note: the incoming records for Sugrue and Crawford were flagged in a DISPLAY statement due to invalid data, and consequently do not appear in the printed report.)

TRUE/FALSE

1. Either OF or IN can qualify a data name.
2. ROUNDED and SIZE ERROR are mandatory in the COMPUTE statement.
3. For the CORRESPONDING option to work, both data names must be at the same level.
4. Qualification over a single level will always remove ambiguity of data names.
5. The same entry may contain both an OCCURS clause and a PICTURE clause.
6. The same entry may not have both an OCCURS clause and a VALUE clause.
7. A single statement cannot have two IFs and one ELSE.
8. Condition names are also known as 77-level entries.
9. The same PERFORM statement can be used to initialize and increment a subscript.
10. The condition portion of an IF may be a compound condition.

EXERCISES

1. How many storage positions are allocated for each of the following table definitions? Show an appropriate schematic indicating storage assignment for each table.

(a) 01 STATE-TABLE.

05 STATE-NAME OCCURS 50 TIMES PIC X(15).

05 STATE-POPULATION OCCURS 50 TIMES PIC 9(8).

(b) 01 STATE-TABLE.

05 NAME-POPULATION OCCURS 50 TIMES.

10 STATE-NAME PIC X(15).

10 STATE-POPULATION PIC 9(8).

2. Company XYZ has four corporate functions: manufacturing, marketing, financial, and administrative. Each function in turn has several departments as shown:

<i>Function</i>	<i>Departments</i>
Manufacturing	10, 12, 16, 17-29, 30, 41, 56
Marketing	6-9, 15, 31-33
Financial	60-62, 75
Administrative	1-4, 78

Establish condition name entries so that given a value of EMPLOYEE-DEPARTMENT one can determine function. Include an 88-level entry, VALID-CODES, to verify that the incoming department is indeed a valid department (any department number not shown is invalid).

3. Given the code:

PROCEDURE DIVISION.

PERFORM SEC-A.

PERFORM PAR-C THRU PAR-E.

MOVE 1 TO N.

PERFORM PAR-G UNTIL N > 3.

STOP RUN.

SEC-A SECTION.

ADD 1 TO X.

ADD 1 TO Y.

ADD 1 TO Z.

PAR-B.

ADD 2 TO X.

PAR-C.

ADD 10 TO X.

PAR-D.

ADD 10 TO Y.

ADD 20 TO Z.

PAR-E.

EXIT.

PAR-F.

MOVE 2 TO N.

PAR-G.

ADD 1 TO N.

ADD 5 TO X.

- (a) How many times is each paragraph executed?
- (b) What is the final value of X, Y, and Z? (Assume they were all initialized to 0.)
- (c) What would happen if the statement ADD 1 TO N were removed from PAR-G?

4. Given the following IF statement:

```

IF A = B OR A = C
  PERFORM FIRST-ROUTINE
ELSE
  IF A > 10 AND B > 10 OR C = 10
    PERFORM SECOND-ROUTINE
  ELSE
    PERFORM THIRD-ROUTINE.

```

For each set of values, indicate which routine will be performed.

- (a) A = 50, B = 5, C = 5
- (b) A = 50, B = 40, C = 40
- (c) A = 50, B = 50, C = 50
- (d) A = 50, B = 5, C = 10
- (e) A = 0, B = 0, C = 0
- (f) A = 1, B = 2, C = 3

5. Given the following Data Division entries and the Procedure Division statement MOVE CORRESPONDING RECORD-ONE TO RECORD-TWO:

```

01 RECORD-ONE.
   05 FIELD-A          PIC X(4).
   05 FIELD-B          PIC X(4).
   05 FIELD-C.
      10 C-ONE         PIC X(4).
      10 C-TWO         PIC X(4).
   05 FIELD-D.
      10 D-ONE         PIC X(6).
      10 D-TWO         PIC X(6).
      10 D-THREE       PIC X(6).
01 RECORD-TWO.
   15 FIELD-E          PIC X(8).
   15 FIELD-D          PIC X(18).
   15 FIELD-C          PIC X(8).
   15 FIELD-B          PIC X(2).
   15 FIELD-A          PIC X(4).
   15 FIELD-F          PIC X(4).
   15 FIELD-G          PIC X(4).
   15 FIELD-H          PIC X(4).

```

Answer true or false (refer to the receiving field):

- (a) The value of FIELD-E is unchanged.
- (b) The value of FIELD-D is unchanged.
- (c) No moves at all will take place since the corresponding level numbers are different in both records.
- (d) The value of FIELD-A will be unchanged since it is the first entry in RECORD-ONE but the fifth entry in RECORD-TWO.
- (e) The value of FIELD-B will be unchanged since the length is different in both records.

6. Consider the following code, intended to calculate an individual's age from a stored birth date and the date of execution.

```

01 EMPLOYEE-RECORD.
   05 EMP-BIRTH-DATE.
       10 BIRTH-MONTH      PIC 99.
       10 BIRTH-YEAR       PIC 99.

01 DATE-WORK-AREA.
   05 TODAYS-MONTH        PIC 99.
   05 TODAYS-DAY          PIC 99.
   05 TODAYS-YEAR         PIC 99.

.
.
PROCEDURE DIVISION.
  ACCEPT DATE-WORK-AREA FROM DATE.
.
.
  COMPUTE EMPLOYEE-AGE = TODAYS-YEAR - BIRTH-YEAR
    + TODAYS-MONTH - BIRTH-MONTH.

```

There are *two* distinct reasons why the code will not work as intended. Find and correct the errors. (Hint: It may be helpful to plug in data and play computer.)

PROJECTS

1. Write a program to print a list of patients seen by a doctor's office for one day. A computerized record has been prepared for each patient with the following format:

<i>Columns</i>	<i>Field</i>	<i>Picture</i>
1-15	LAST-NAME	X(15)
16-25	FIRST-NAME	X(10)
26-50	REASON-FOR-SEEING-DOCTOR	X(25)
51-55	AMOUNT-PAID	9(3)V99

The following items should be considered in developing your program:

- The amount paid has spaces in the high-order positions. Use the INSPECT verb to convert the leading spaces to zeros.
- Use the same data names in the input file record layout as in the print file layout. Use one MOVE CORRESPONDING to move the input fields to the print line.
- Obtain today's date through an ACCEPT statement and move it to a heading line.
- Keep a count of the number of patients seen and a total of the amount paid. Print these on the total line.
- Use the COMPUTE verb with the ROUNDED option to calculate the cost of an average visit (i.e., the TOTAL PAID divided by the number of patients).
- Use double spacing before each detail line. Allow only five detail lines per page.
- Develop your own test data in a *separate* file.

2. A television store in a large city guarantees that its antennas will stay on residents' roofs at least six years or else they will reinstall the antenna free. Customer records have been entered into a data file with the following information:

<i>Columns</i>	<i>Field</i>	<i>Picture</i>
1-25	NAME	X(25)
26-50	ADDRESS	X(25)
51-52	YEAR-OF-INSTALLATION	99
53-54	MONTH-OF-INSTALLATION	99
58-60	TYPE-OF-ANTENNA	XXX

Write a program that will print only the names of customers whose antennas were installed within the last six years. Put an asterisk next to the name of any customer whose antenna is at least four years old. Define your input record as PIC X(60) and your print record as X(132). Define a work record for the input file in Working-Storage and three work records for the print file (for heading, detail, and total lines). Keep a count of the number of guarantees in effect. Use READ INTO and WRITE FROM. Use duplicate names in at least three fields of the detail print line and input records in Working-Storage. Use a MOVE CORRESPONDING when moving the input fields to the print record. Use an 88-level entry to define a switch to determine end of file.

Develop your own test data.

7

**PROGRAMMING
STYLE**

OVERVIEW As a beginning programmer, your objective should simply be a working program. As a professional, your objective is enlarged, namely, a program which is easily read and maintained by someone other than yourself. A good program must also be tested under a variety of conditions, including obviously improper data, and should include programming checks to flag potentially invalid transactions. In short, while the beginner is concerned with merely translating a working flowchart or pseudocode into COBOL, the professional requires a better flowchart, straightforward logic, easy-to-read COBOL code, and a well-tested program.

Over time, an individual develops a collection of techniques, i.e., a style, to accomplish his or her objectives. Most books omit programming style entirely, or at best devote only a few pages to the subject. We believe the topic is so important that it merits an entire chapter. We have found that an awareness of "style" is highly beneficial to student and professional alike. Individuals conscious of style tend to write programs that are easier to read and maintain, easier to debug, and more apt to be correct.

This chapter is divided into two main sections: *coding standards* and *structured programming*. The first deals with "dos and don'ts" of COBOL, and contains a set of 12 guidelines for writing better programs. The second section defines structured programming, and emphasizes the *functional* nature of COBOL paragraphs. The unit concludes with a substantial modification of the payroll program of Chapter 4, with special attention to programming style.

CODING STANDARDS In spite of what you may think of the COBOL compiler, COBOL is a relatively free-form language. There is considerable flexibility as to starting column for most entries (i.e., in or beyond column 12). The rules for paragraph and data names make for easy-to-write, but not necessarily easy-to-read programs.

In a business situation, it is absolutely essential that programs be well documented, as the person who writes a program today may not be here tomorrow. Indeed, *continuing success* depends on someone other than the author being able to maintain a program. Accordingly, most installations (both large and small) impose a set of coding standards, which go beyond the requirements of COBOL. Such standards are optional for the student. However, they are typical of what is required in the real world.

Some of the guidelines we discuss are purely cosmetic and are concerned only with the arrangement of the COBOL code. Indentation, spacing, avoiding commas, and so on fall into this class. Other guidelines suggest particular COBOL features to use and/or avoid; for example, performing paragraphs rather than sections, using 88-level entries, avoiding 77-level entries, and so on.

The suggestions in this section should be viewed as guidelines rather than rigid standards. Consequently, the reader is not necessarily expected to agree with all items. If you have sound reason for objecting to an element of the style presented here, so be it. You are on your way to developing your own.

Let us list the elements of programming style which we will consider:

- Choose meaningful names
- Avoid commas

- Use appropriate comments
- Eliminate 77-level entries
- Space attractively
- Indent
- Avoid constants
- Avoid literals
- Keep it simple
- Perform paragraphs, not sections
- Restrict subscripts to a single use
- Use 88-level entries

CHOOSE MEANINGFUL NAMES

The COBOL compiler is very lenient with its rules for programmer-chosen names. Specifically, a user-defined word may not exceed 30 characters, or begin or end with a hyphen. Valid characters for inclusion in a user-defined word are A through Z, 0 through 9, and the hyphen. File and data names must contain at least one alphabetic character, whereas paragraph and section names may be all numeric. It is strongly recommended that these rules be amended as follows:

1. Data names should be mnemonically significant. Although COBOL allows up to 30 characters, two- and three-character cryptic names are used too frequently. It is impossible for the maintenance programmer, or even the original author, to determine the meaning of abbreviated data names. On first reading, it may seem that this guideline adds unnecessarily to the burden of “writer’s cramp”. Experience has shown, however, that meaningful data names significantly ease the job of the maintenance programmer. Some examples:

Poor Choice:

```
SWITCH-ONE
TOTAL-1
TRANS-ID
```

Improved Choice:

```
END-OF-TRANSACTION-FILE-SWITCH
TOTAL-EMPLOYEE-GROSS-PAY
TRANSACTION-ID-NUMBER
```

2. All data names within the same 01 record should have a common two- or three-letter prefix. The utility of this guideline becomes apparent in the Procedure Division if it is necessary to refer back to the definition of a data name. Some examples:

Poor Choice:

```
01 EMPLOYEE-RECORD.
   05 SOC-SEC-NUMBER          PIC 9(9).
   05 NAME                    PIC X(20).
   05 ADDRESS                  PIC X(40).
```

Improved Choice:

```

01 EMPLOYEE-RECORD.
   05 EMP-SOC-SEC-NUMBER      PIC 9(9).
   05 EMP-NAME                PIC X(20).
   05 EMP-ADDRESS             PIC X(40).

```

3. Paragraph names should be *functional* and *reflect the single purpose of the paragraph*. A paragraph name should consist of a verb, an adjective or two, and an object, e.g., READ-TRANSACTION-FILE, ADD-NEW-RECORD, and so on. If a paragraph cannot be named in this manner, it is probably not functional, and consideration should be given to redesigning the program and/or paragraph. Paragraph names should also be sequenced to locate paragraphs quickly in the Procedure Division. There is, however, considerable disagreement on just what sequencing scheme to use; e.g., all numbers, a single letter followed by numbers, etc. This author will make no strong argument for one scheme over another, other than to insist that a consistent sequencing rule be followed. Some examples:

Poor Choice:

```

0005-MAINLINE
A010-READ-AND-WRITE
READ-TRANSACTION-FILE

```

Improved Choice:

```

A010-WRITE-NEW-MASTER-RECORD
1000-PRODUCE-ERROR-REPORT
2000-READ-TRANSACTION-FILE

```

AVOID COMMAS The compiler treats a comma as “noise”; i.e., a comma has no effect on the generated code. Many programmers have acquired the habit of inserting commas to increase readability. While this works rather well with prose, it can have just the opposite effect in COBOL. This is because of blurred print chains which make it difficult to distinguish a comma from a period. As we have already seen, the presence or absence of a period is critical, and the inability to distinguish a period from a comma becomes rather annoying; consequently, try avoiding commas altogether.

USE APPROPRIATE COMMENTS Although there is growing disillusionment with comments in COBOL programs, good code does not eliminate their necessity. As Yourdon[†] has so eloquently stated, “no programmer, no matter how wise, no matter how experienced, no matter how hard pressed for time, no matter how well intentioned, should be forgiven an uncommented and undocumented program.” The mere presence of comments, however, does not insure a well-documented program, and poor comments are sometimes worse than no

[†]Edward Yourdon, *Techniques of Program Structure and Design*, Prentice-Hall, Inc., 1975.

comments at all. The most common fault is *redundance* with the source code. For example, in the code:

```
*** CALCULATE NET PAY
    COMPUTE NET-PAY = GROSS-PAY - FEDERAL-TAX - VOLUNTARY-DEDUCTION.
```

the comment does not add to the readability of the program. It might even be said to detract from legibility because it breaks the logical flow as one is reading. Worse than redundant, comments may be obsolete or incorrect; i.e., inconsistent with the associated code. This happens if program statements are changed during debugging or maintenance, and the comments are not correspondingly altered. The compiler, unfortunately, does not validate comments. Comments may also be correct, but incomplete and hence misleading. In sum, the presence of comments is essential, but great care, *more than is commonly exercised*, should be applied in developing and maintaining comments in a program.

As a general rule, comments should be provided whenever you are doing something which is not immediately obvious to another person. When considering a comment, imagine you are turning the program over for maintenance, and insert comments whenever you would pause to explain a feature in your program. Do assume, however, that the maintenance programmer is as competent in COBOL as you are. Thus, comments should be directed to *why* you are doing something, rather than to what you are doing.

ELIMINATE 77-LEVEL ENTRIES 77-level entries were originally conceived as being independent items with no relationship to one another. In reality few, if any, data names are truly independent, and the alternative is to group *related* entries under a *common* 01 description in Working-Storage. Consider:

Poor Code:

```
77 COUNTER-ONE      PIC 9(3)  VALUE ZEROS.
77 COUNTER-TWO      PIC 9(3)  VALUE ZEROS.
77 COUNTER-THREE    PIC 9(3)  VALUE ZEROS.
```

Improved Code:

```
01 RECORD-COUNTERS.
   05 NUMBER-OF-RECORDS-READ  PIC 9(3)  VALUE ZEROS.
   05 NUMBER-OF-GOOD-RECORDS  PIC 9(3)  VALUE ZEROS.
   05 NUMBER-OF-BAD-RECORDS   PIC 9(3)  VALUE ZEROS.
```

The improved code has given the three counters more descriptive names which reflect both similarities and differences among the related items.

SPACE ATTRACTIVELY The adoption of various spacing conventions can go a long way toward improving the appearance and legibility of a program. This author believes very strongly in the insertion of blank lines throughout a program to highlight important statements. Specific suggestions include a blank line before all paragraph and/or section headers, before all FDs and/or 01 entries, and even before specific verbs, e.g., IF.

The reader can also cause various portions of a listing to begin on a new page, e.g., division headers. This is accomplished by putting a slash in column 7 of a source statement.

Vertical spacing is also important. The Data Division, for example, is enhanced significantly by beginning all picture clauses in the same column.

INDENT Virtually no one will argue against indenting successive level numbers within a record description in the Data Division. Why then do so few employ indentation in the Procedure Division? Consider:

Poor Code:

```
PERFORM INITIALIZE-TABLE VARYING LOCATION-SUB FROM 1
BY 1 UNTIL LOCATION-SUB > 3.

READ EMPLOYEE-FILE AT END MOVE "YES" TO END-EMPLOYEE-SWITCH.

WRITE PRINT-LINE AFTER ADVANCING PAGE.

IF EMPLOYEE-AGE > 65 MOVE EMP-NAME TO
PRINT-RETIREMENT-NAME, ADD 1 TO
NUMBER-OF-RETIRES, PERFORM WRITE-RETIREE-REPORT.
```

Improved Code:

```
PERFORM INITIALIZE-TABLE
    VARYING LOCATION-SUB FROM 1 BY 1
    UNTIL LOCATION-SUB > 3.

READ EMPLOYEE-FILE
    AT END MOVE "YES" TO END-EMPLOYEE-SWITCH.

WRITE PRINT-LINE
    AFTER ADVANCING PAGE.

IF EMPLOYEE-AGE > 65
    MOVE EMP-NAME TO PRINT-RETIREMENT-NAME
    ADD 1 TO NUMBER-OF-RETIRES
    PERFORM WRITE-RETIREE-REPORT.
```

As can be seen from the improved code, subservient clauses should always be indented under the main verbs. The legibility of PERFORM, for example, is improved immeasurably by indenting VARYING under PERFORM, and UNTIL under VARYING. Other examples in the same vein include:

```
and      AFTER (BEFORE) ADVANCING under WRITE,
        AT END under READ,
        SIZE ERROR under COMPUTE,
        GIVING under ADD, MULTIPLY, SUBTRACT, and DIVIDE.
```

Indentation should also be *consistent* with compiler interpretation. The IF statement, for example, is terminated by a period and the indentation should reflect this. The condition portion is written on a line by itself in the

preceding improved code with the subservient statements (MOVE, ADD, and PERFORM) indented under it.

The *nested* IF statement is worthy of special mention. The compiler does not interpret ELSE clauses as the programmer writes them but *associates the ELSE clause with the closest unpaired previous IF*. Consider:

Poor Code:

```
IF CD-SEX IS EQUAL TO "M"
  IF CD-AGE IS GREATER THAN 30
    MOVE CD-NAME TO MALE-OVER-30
    ADD 1 TO NUMBER-QUALIFIED-MALES
  ELSE MOVE CD-NAME TO PRT-NAME
  ADD 1 TO MALE-UNDER-30.
```

The indentation implies that CD-NAME will be moved to PRT-NAME if CD-SEX is not equal to "M". This is *not* the compiler interpretation. The ELSE clause is associated with the closest previous IF which is not already paired with another ELSE. Therefore, the compiler will move CD-NAME to PRT-NAME if CD-SEX equals "M" but CD-AGE is not greater than 30.

Nested IFs should be coded as follows:

1. Indent successive IFs four columns.
2. Put the word ELSE on a line by itself, and directly under its associated IF.
3. Indent detail lines for both IF and ELSE four columns.

The previous nested IF statement is rewritten to reflect these guidelines:

Improved Code:

```
IF CD-SEX IS EQUAL TO "M"
  IF CD-AGE IS GREATER THAN 30
    MOVE CD-NAME TO MALE-OVER-30
    ADD 1 TO NUMBER-QUALIFIED-MALES
  ELSE
    MOVE CD-NAME TO PRT-NAME
    ADD 1 TO MALE-UNDER-30.
```

AVOID CONSTANTS

A significant portion of maintenance programming (and headaches) could probably be avoided if the original program were written with an eye toward future change. Consider:

Poor Code:

```
05 STATE-TABLE OCCURS 50 TIMES.
10 STATE-POPULATION      PIC 9(8).
10 STATE-NAME            PIC X(15).
.
.
PERFORM COMPUTE-STATE-TOTALS
  VARYING STATE-SUBSCRIPT FROM 1 BY 1
  UNTIL STATE-SUBSCRIPT > 50.

COMPUTE AVERAGE-STATE-POPULATION = TOTAL-POPULATION / 50.
```

Improved Code:

```

05  NUMBER-OF-STATES      PIC 99  VALUE 50.
.
.
05  STATE-TABLE OCCURS 55 TIMES.
    10  STATE-POPULATION  PIC 9(8).
    10  STATE-NAME        PIC X(15).
.
.
    PERFORM COMPUTE-STATE-TOTALS
        VARYING STATE-SUBSCRIPT FROM 1 BY 1
        UNTIL STATE-SUBSCRIPT > NUMBER-OF-STATES.

    COMPUTE AVERAGE-STATE-POPULATION =
        TOTAL-POPULATION / NUMBER-OF-STATES.

```

Admittedly, it has been some 20 years since Alaska and Hawaii became states. Nevertheless, if and when another state is admitted, the improved code is decidedly easier to modify. All that needs to be changed is the value of NUMBER-OF-STATES. The poor code, however, requires changes in several places; specifically, the constant 50 has to be changed to 51 three times. The possibility of error is much greater, as the programmer is required to track down all instances where the value changes. Constants do not appear on a cross reference listing.

A second benefit of avoiding constants in favor of variable data names is increased readability. Consider:

Poor Code:

```
ADD .04 .04 GIVING SALES-TAX-PERCENTAGE.
```

Improved Code:

```
ADD NEW-YORK-STATE-SALES-TAX
NEW-YORK-CITY-SALES-TAX
GIVING SALES-TAX-PERCENTAGE.
```

The reader is hard pressed to determine the meaning of either occurrence of .04 in the first example, whereas the meaning is obvious in the second example. True, the latter requires definition of additional data names in the Data Division and extra pencil strokes in the Procedure Division. This is a small price to pay, however, for the increased legibility and ease of maintenance.

AVOID LITERALS The constant (literal) portion of a print line should be defined in Working-Storage, rather than being moved to the print line in the Procedure Division. Consider:

Poor Code:

```

MOVE "STUDENT-NAME      SOC SEC NUM  CREDITS  TUITION
    "SCHOLARSHIP FEES" TO PRINT-LINE.
WRITE PRINT-LINE.

```

Hyphen required to continue non-numeric literal

Improved Code:

```

01  HEADING-LINE.
    05  FILLER          PIC X(12)  VALUE "STUDENT NAME".
    05  FILLER          PIC X(10)  VALUE SPACES.
    05  FILLER          PIC X(11)  VALUE "SOC SEC NUM".
    05  FILLER          PIC X(2)   VALUE SPACES.
    05  FILLER          PIC X(7)   VALUE "CREDITS".
    05  FILLER          PIC X(2)   VALUE SPACES.
    05  FILLER          PIC X(7)   VALUE "TUITION".
    05  FILLER          PIC X(3)   VALUE SPACES.
    05  FILLER          PIC X(11)  VALUE "SCHOLARSHIP".
    05  FILLER          PIC X(2)   VALUE SPACES.
    05  FILLER          PIC X(4)   VALUE "FEES".
    05  FILLER          PIC X(61)  VALUE SPACES.

WRITE PRINT-LINE FROM HEADING-LINE.

```

The improved code may appear unnecessarily long in contrast to the poor code. However, it is an unwritten law that users will change column headings and/or spacing at least twice before being satisfied. Such changes are easily accommodated in the improved code, but often tedious in the alternative solution. Assume, for example, that four spaces are required between CREDITS and TUITION, rather than the two that are there now. Modification of the poor code requires that *both* lines in the MOVE statement be completely rewritten, whereas only a single PICTURE clause need be changed in the improved version.

**KEEP IT
SIMPLE**

Procedure Division code should be kept as straightforward as possible, and efforts at being cute or fancy should be discouraged. Beginning programmers, especially, are notorious for trying to impress their peers with 'clever' code, which too often confuses the issue.

Consider the following payroll specification for hourly employees: all employees receive straight time for the first 40 hours worked, time and a half for the next 8 hours, and double time for any hours over 48. For example, an employee who worked 50 hours with an hourly rate of \$5.00 should receive \$280.00 (40 hours at \$5.00, 8 hours at \$7.50, and 2 hours at \$10.00).

The following, *logically equivalent* IF statements, are partial solutions:

```

IF HOURS-WORKED > 48
  COMPUTE GROSS-PAY
    = 40 * HOURLY-RATE
    + 8 * HOURLY-RATE * 1.5
    + (HOURS-WORKED - 48) * HOURLY-RATE * 2.

IF HOURS-WORKED > 48
  COMPUTE GROSS-PAY
    = 52 * HOURLY-RATE
    + (HOURS-WORKED - 48) * HOURLY-RATE * 2.

```

The first statement is a line longer, but is the preferred solution as it more closely represents the physical problem. It is easy to see that individuals working more than 48 hours receive straight time for the first 40 hours, time and a half for the next 8 hours, and double time for any hours over 48. Although the second statement produces equivalent results, it deviates sig-

nificantly from the physical situation. A maintenance programmer would be hard pressed to understand the meaning of the constant 52. The second statement may be more elegant in a mathematical sense, but it is certainly undesirable in a commercial environment.

PERFORM PARAGRAPHS, NOT SECTIONS

The motivation behind this guideline is best demonstrated by example. Given the following Procedure Division, what will be the final value of X?

```
PROCEDURE DIVISION.
MAINLINE SECTION.
    MOVE ZEROS TO X.
    PERFORM A.
    PERFORM B.
    PERFORM C.
    PERFORM D.
    STOP RUN.
A SECTION.
    ADD 1 TO X.
B.
    ADD 1 TO X.
C.
    ADD 1 TO X.
D.
    ADD 1 TO X.
```

The *correct* answer is 7, *not* 4. A common error made by many programmers is a misinterpretation of the statement PERFORM A. *Since A is a section and not a paragraph*, the statement PERFORM A invokes *every* paragraph in that section, namely, paragraphs B, C, and D, in addition to the unnamed paragraph immediately after the section header.

The PERFORM statement specifies a *procedure*, which can be *either* a section or a paragraph. Unfortunately, there is no way of telling the nature of the procedure from the PERFORM statement itself. Consequently, when a section is specified as a procedure, the unfortunate result is too often execution of unintended code. Can't happen? Did you correctly compute the value of X?

RESTRICT SUBSCRIPTS TO A SINGLE USE

Data names defined as switches and/or subscripts should be restricted to a single use. Consider:

Poor Code:

```
77 SUBSCRIPT    PIC 9(4).
.
.
PERFORM COMPUTE-SALARY-HISTORY
    VARYING SUBSCRIPT FROM 1 BY 1
    UNTIL SUBSCRIPT > 3.

PERFORM FIND-MATCHING-TITLE
    VARYING SUBSCRIPT FROM 1 BY 1
    UNTIL SUBSCRIPT > 100.
```

Improved Code:

```

01 PROGRAM-SUBSCRIPTS.
   05 TITLE-SUBSCRIPT      PIC 9(4).
   05 SALARY-SUBSCRIPT     PIC 9(4).

   PERFORM COMPUTE-SALARY-HISTORY
       VARYING SALARY-SUBSCRIPT FROM 1 BY 1
       UNTIL SALARY-SUBSCRIPT > 3.

   PERFORM FIND-MATCHING-TITLE
       VARYING TITLE-SUBSCRIPT FROM 1 BY 1
       UNTIL TITLE-SUBSCRIPT > 100.

```

At the very least, the improved code offers superior documentation. By restricting data names to a single use, one automatically avoids such non-descript entries as SUBSCRIPT. Of greater impact, the improved code is more apt to be correct in that a given data name is modified or tested in fewer places within a program. Finally, if bugs do occur, the final values of the unique data names (e.g., TITLE-SUBSCRIPT and SALARY-SUBSCRIPT) will be of much greater use than the single value of SUBSCRIPT.

USE 88-LEVEL ENTRIES

Condition names (88-level entries) are useful to improve documentation. They facilitate program change and can reduce the need for compound conditions in an IF statement. Consider the case of a political candidate seeking the names of registered Democrats in two Florida cities:

Poor Code:

```

IF (LOCATION-CODE = 48 OR LOCATION-CODE = 65)
  AND POLITICAL-PARTY = "D" ...

```

Improved Code:

```

05 LOCATION-CODE          PIC 99.
   88 MIAMI                VALUE 48.
   88 TAMPA                VALUE 65.
   88 FLORIDA              VALUES ARE 48, 65.

05 POLITICAL-PARTY        PIC X.
   88 DEMOCRAT             VALUE "D".
   88 REPUBLICAN            VALUE "R".

IF FLORIDA AND DEMOCRAT ...

```

When 88-level entries are *not* used, the IF statement is considerably harder to read. Moreover, the chances for error are greater as the condition portion is more complex to code. In the example shown, the parentheses are required, and the meaning will change if they are removed. The improved code defines the city and political party codes in the Data Division, resulting in an easier to read Procedure Division.

Condition names also facilitate maintenance in that changes to existing codes, and/or additions of new codes is done in only one place, the Data Division. When 88-level entries are not used, and the value of a given data

name is tested more than once in the Procedure Division, changes are required in several places.

STRUCTURED PROGRAMMING

Although coding standards are essential to a well-written program, they do not by themselves guarantee success. Attention must also be focused on logic, and its implementation in a program.

This is the first discussion of the term, *structured programming*, although every program presented so far has been “structured.” Structured programming is the discipline of making a program’s logic easy to follow. This is accomplished by limiting a program’s flowchart to three basic building blocks: *sequence*, *selection*, and *iteration*, as shown in Figure 7.1.

Sequence specifies that statements in a program are executed sequentially, that is, in the order in which they appear. *Selection* is the choice between two actions. A condition is tested; if it is true, block A is executed; if

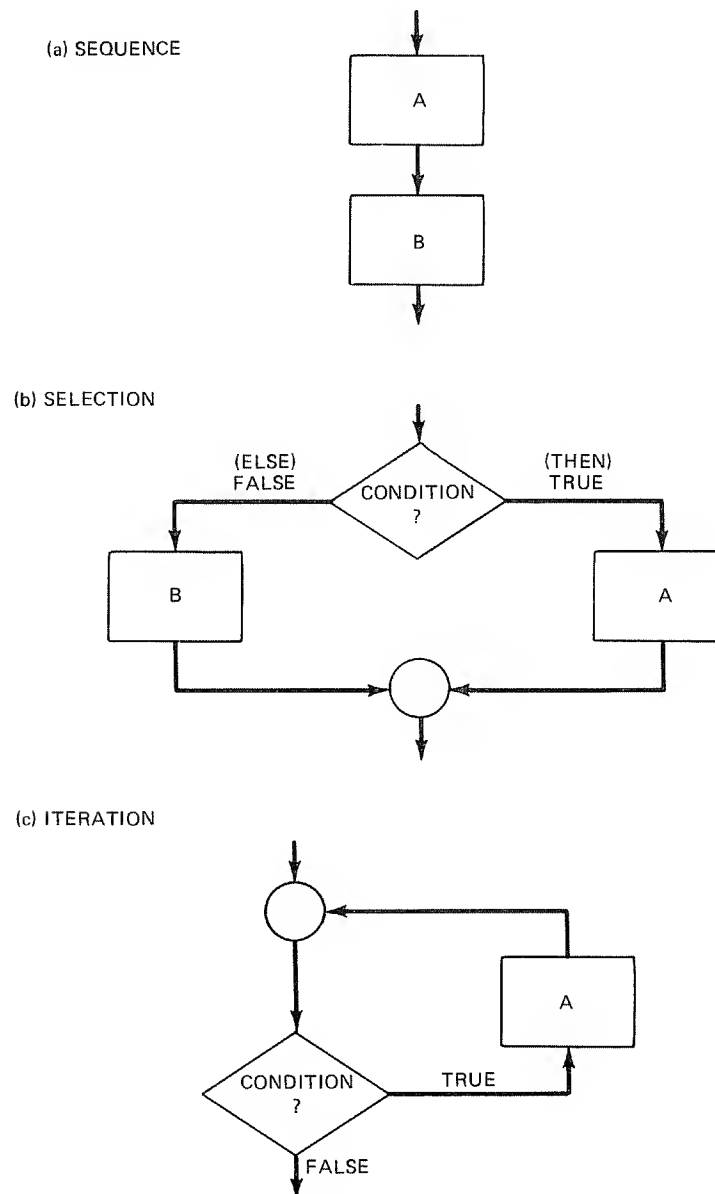


FIGURE 7.1 Building blocks of structured programming

it is false, block B is executed. *Iteration* (or looping) calls for repeated execution of one or more instructions while a condition remains true.

The theory of structured programming is easily applied to COBOL. The sequence structure is implemented by coding statements in order. Selection is implemented by the IF/ELSE statement, and iteration through the PERFORM/UNTIL.

Conspicuous by its absence in Figure 7.1 is the unconditional transfer of control, or GO TO statement. The originators of structured programming did not deliberately avoid the GO TO, but rather saw no need for its use. The three logic structures of Figure 7.1 are sufficient to produce any required logic; indeed, we have come all the way through the book without using this seemingly innocuous statement.

The happy result of the structured discipline is that programs written this way are easier to read and more apt to be correct. This will become very evident in later chapters as we progress to programs requiring more complex logic. Structured programming also requires that every paragraph in a COBOL program do one and only one job. The utility of this principle is best illustrated by example. Recall the specifications of the original payroll program of Chapter 4:

1. Process a file of employee records, calculating regular, overtime, and gross pay. Print a detail line for every employee.
2. Maintain company totals for regular, overtime, and gross pay, and print these three numbers at the end of the report.
3. Print a simple heading line at the beginning of the report.

The structure of the original payroll program was shown in the hierarchy chart of Figure 4.7. The paragraph, PREPARE-PAYROLL, was on top of the chart and called the subordinate paragraphs WRITE-HEADING-LINE, PROCESS-RECORDS, and WRITE-COMPANY-TOTAL. PROCESS-RECORDS, in turn, called two lower level modules, UPDATE-COMPANY-TOTALS and WRITE-DETAIL-LINE.

We now expand the payroll program to accommodate the following changes, and in the process illustrate the structured design principle of *functional* paragraphs:

1. A maximum of three employees is to appear on each page of the report. Consequently, the heading line is to be printed an indeterminate number of times (once on each page) rather than only at the beginning, as was done in Chapter 4. In addition, a second heading line containing a page number and date of execution is to be added.
2. Federal tax is to be computed for each employee and deducted from gross pay. Federal tax is calculated as follows:

16% of the first \$160
18% on amounts between \$160 and \$200
20% on anything over \$200

Thus, an employee earning \$275 would pay \$47.80 in tax as follows:

16% of \$160	= \$25.60
18% on \$ 40	= \$ 7.20
20% on \$ 75	= \$15.00
	<hr/>
	Tax = \$47.80

3. The heading, detail, and total lines are to be modified to accommodate both federal tax and net pay. (Realize that company totals must also be maintained for these items.)

A hierarchy chart for the expanded payroll program is shown in Figure 7.2. Note well the strong *functional* nature (verb, adjective, object) of each box in the chart. The "job" of each module is readily apparent, and further is restricted to a *single* function. The hierarchy chart does not contain decision making logic as does pseudocode or a flowchart. It does, however, imply the specific requirements of each paragraph in the program, and consequently is a valuable aid in both design and documentation.

THE COMPLETED PROGRAM

Figure 7.3 contains the completed program. It is fully structured, and adheres to several of the coding guidelines discussed earlier. Blank lines appear before 01 entries and paragraph names. All PICTURE and VALUE clauses are vertically aligned.

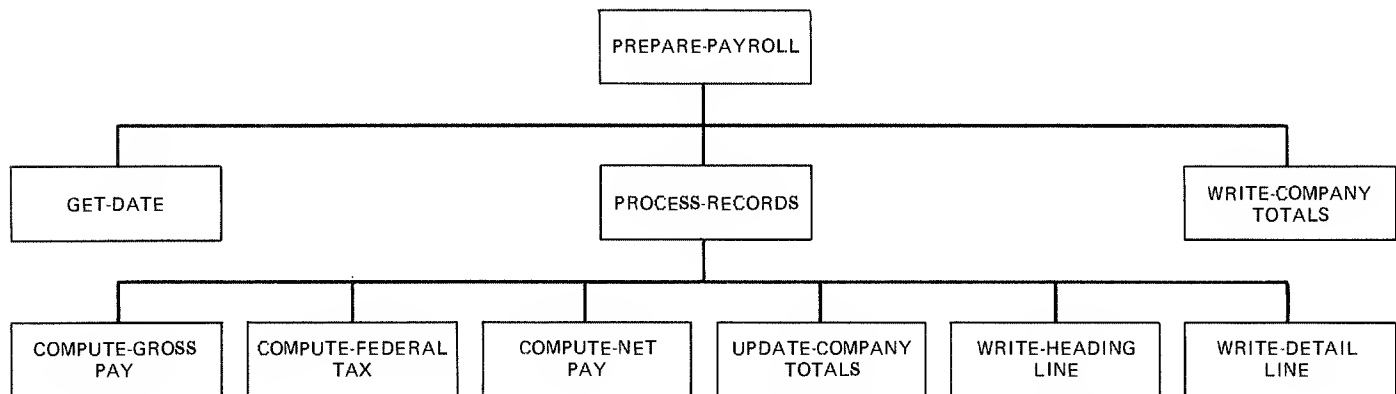


FIGURE 7.2 Hierarchy chart for expanded payroll program

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID. PAYROLL2.
000120 AUTHOR. R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER. TRS-80.
000170 OBJECT-COMPUTER. TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210 SELECT EMPLOYEE-FILE
000220 ASSIGN TO INPUT "PAYROLL/DAT".
000230 SELECT PRINT-FILE
000240 ASSIGN TO PRINT "PAYROLL2/TXT".
000250
000260 DATA DIVISION.
000270 FILE SECTION.
000280 FD EMPLOYEE-FILE
000290 LABEL RECORDS ARE OMITTED
000300 RECORD CONTAINS 80 CHARACTERS
000310 DATA RECORD IS EMPLOYEE-RECORD.
000320 01 EMPLOYEE-RECORD.
000330 05 EMP-NAME.
000340 10 EMP-LAST-NAME PIC X(15).
000350 10 EMP-FIRST-NAME PIC X(10).
  
```

Data names within an FD have common prefix

FIGURE 7.3 Expanded payroll program

000360	05	EMP-HOURS-WORKED.		
000370	10	EMP-REG-HOURS	PIC 99.	
000380	10	EMP-OVERTIME-HOURS	PIC 99.	
000390	05	EMP-RATE	PIC 99V99.	
000400	05	FILLER	PIC X(47).	
000410				
000420	FD	PRINT-FILE		
000430		LABEL RECORDS ARE STANDARD		
000440		RECORD CONTAINS 132 CHARACTERS		
000450		DATA RECORD IS PRINT-LINE.		
000460	01	PRINT-LINE	PIC X(132).	
000470				
000480		WORKING-STORAGE SECTION.		
000490	77	WS-DATA-REMAINS-SWITCH	PIC X(3)	VALUE SPACES.
000500				
000510	01	DATE-WORK-AREA.		Holds date of execution
000520	05	TODAYS-YEAR	PIC 99.	
000530	05	TODAYS-MONTH	PIC 99.	
000540	05	TODAYS-DAY	PIC 99.	
000550				
000560	01	IND-COMPUTATIONS.		Common prefix
000570	05	IND-REGULAR-PAY	PIC 9(4)V99.	
000580	05	IND-OVERTIME-PAY	PIC 9(4)V99.	
000590	05	IND-GROSS-PAY	PIC 9(4)V99.	
000600	05	IND-FEDERAL-TAX	PIC 9(4)V99.	
000610	05	IND-NET-PAY	PIC 9(4)V99.	
000620				
000630	01	COMPANY-TOTALS.		PICTURE and VALUE clauses are vertically aligned
000640	05	CO-REGULAR-PAY	PIC 9(6)V99	VALUE ZEROS.
000650	05	CO-OVERTIME-PAY	PIC 9(6)V99	VALUE ZEROS.
000660	05	CO-GROSS-PAY	PIC 9(6)V99	VALUE ZEROS.
000670	05	CO-FEDERAL-TAX	PIC 9(6)V99	VALUE ZEROS.
000680	05	CO-NET-PAY	PIC 9(6)V99	VALUE ZEROS.
000690				
000700	01	PAGE-AND-LINE-COUNTERS.		
000710	05	WS-PAGE-COUNT	PIC 9(4)	VALUE ZEROS.
000720	05	WS-LINE-COUNT	PIC 9(4)	VALUE 4.
000730				Page and line counters are required for page heading routine
000740	01	HEADING-LINE-ONE.		
000750	05	FILLER	PIC X(4).	
000760	05	HDG-MONTH	PIC Z9.	
000770	05	FILLER	PIC X	VALUE "/".
000780	05	HDG-DAY	PIC Z9.	
000790	05	FILLER	PIC X	VALUE "/".
000800	05	HDG-YEAR	PIC 99.	
000810	05	FILLER	PIC X(40)	VALUE SPACES.
000820	05	FILLER	PIC X(8)	VALUE "PAYROLL".
000830	05	FILLER	PIC X(6)	VALUE "REPORT".
000840	05	FILLER	PIC X(40)	VALUE SPACES.
000850	05	FILLER	PIC X(4)	VALUE "PAGE".
000860	05	HDG-PAGE-NUMBER	PIC Z(4).	
000870	05	FILLER	PIC X(18)	VALUE SPACES.
000880				
000890	01	HEADING-LINE-TWO.		
000900	05	FILLER	PIC X(8)	VALUE SPACES.
000910	05	FILLER	PIC X(4)	VALUE "NAME".
000920	05	FILLER	PIC X(9)	VALUE SPACES.
000930	05	FILLER	PIC X(4)	VALUE "RATE".
000940	05	FILLER	PIC X(4)	VALUE SPACES.
000950	05	FILLER	PIC X(9)	VALUE "REG HOURS".
000960	05	FILLER	PIC X(4)	VALUE SPACES.
000970	05	FILLER	PIC X(9)	VALUE "O/T HOURS".
000980	05	FILLER	PIC X(4)	VALUE SPACES.
000990	05	FILLER	PIC X(7)	VALUE "REG PAY".
001000	05	FILLER	PIC X(4)	VALUE SPACES.
001010	05	FILLER	PIC X(7)	VALUE "O/T PAY".
001020	05	FILLER	PIC X(4)	VALUE SPACES.
001030	05	FILLER	PIC X(11)	VALUE "GROSS PAY".
001040	05	FILLER	PIC X(2)	VALUE SPACES.

FIGURE 7.3 Continued

001050	05	FILLER	PIC X(7)	VALUE "FED TAX".	
001060	05	FILLER	PIC X(5)	VALUE SPACES.	
001070	05	FILLER	PIC X(7)	VALUE "NET PAY".	
001080	05	FILLER	PIC X(23)	VALUE SPACES.	
001090					
001100	01	DASHED-LINE.			Blank lines are inserted before 01 entries
001110	05	ROW-OF-DASHES	PIC X(111)	VALUE ALL "-".	
001120	05	FILLER	PIC X(21)	VALUE SPACES.	
001130					
001140	01	DETAIL-LINE.			
001150	05	FILLER	PIC X(2).		
001160	05	DET-LAST-NAME	PIC X(15).		
001170	05	FILLER	PIC X(2).		
001180	05	DET-RATE	PIC \$\$\$9.99.		
001190	05	FILLER	PIC X(8).		
001200	05	DET-REG-HOURS	PIC Z9.		
001210	05	FILLER	PIC X(10).		
001220	05	DET-OVERTIME-HOURS	PIC Z9.		Datanames within an 01 entry have a common prefix
001230	05	FILLER	PIC X(6).		
001240	05	DET-REGULAR-PAY	PIC \$\$,\$\$9.99.		
001250	05	FILLER	PIC X(3).		
001260	05	DET-OVERTIME-PAY	PIC \$\$,\$\$9.99.		
001270	05	FILLER	PIC X(2).		
001280	05	DET-GROSS-PAY	PIC \$\$,\$\$9.99.		
001290	05	FILLER	PIC X(3).		
001300	05	DET-FEDERAL-TAX	PIC \$\$,\$\$9.99.		
001310	05	FILLER	PIC X(3).		
001320	05	DET-NET-PAY	PIC \$\$,\$\$9.99.		
001330	05	FILLER	PIC X(23).		
001340					
001350	01	TOTAL-LINE.			
001360	05	FILLER	PIC X(6)	VALUE SPACES.	
001370	05	FILLER	PIC X(6)	VALUE "TOTALS".	
001380	05	FILLER	PIC X(41)	VALUE SPACES.	
001390	05	TOTAL-REGULAR-PAY	PIC \$\$,\$\$9.99.		
001400	05	FILLER	PIC X(3)	VALUE SPACES.	
001410	05	TOTAL-OVERTIME-PAY	PIC \$\$,\$\$9.99.		
001420	05	FILLER	PIC X(2)	VALUE SPACES.	
001430	05	TOTAL-GROSS-PAY	PIC \$\$,\$\$9.99.		
001440	05	FILLER	PIC X(3)	VALUE SPACES.	
001450	05	TOTAL-FEDERAL-TAX	PIC \$\$,\$\$9.99.		
001460	05	FILLER	PIC X(3)	VALUE SPACES.	
001470	05	TOTAL-NET-PAY	PIC \$\$,\$\$9.99.		
001480	05	FILLER	PIC X(47)	VALUE SPACES.	
001490					
001500		PROCEDURE DIVISION.			
001510	0100	PREPARE-PAYROLL.			
001520		PERFORM 0200-GET-DATE.			
001530		OPEN INPUT EMPLOYEE-FILE			
001540		OUTPUT PRINT-FILE.			
001550		READ EMPLOYEE-FILE			
001560		AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.			
001570		PERFORM 0300-PROCESS-RECORDS			
001580		UNTIL WS-DATA-REMAINS-SWITCH = "NO".			
001590		PERFORM 1000-WRITE-COMPANY-TOTALS.			
001600		CLOSE EMPLOYEE-FILE			
001610		PRINT-FILE.			
001620		STOP RUN.			
001630					
001640	0200	GET-DATE.			ACCEPT statement to obtain date of execution
001650		ACCEPT DATE-WORK-AREA FROM DATE.			
001660		MOVE TODAYS-YEAR TO HDG-YEAR.			
001670		MOVE TODAYS-MONTH TO HDG-MONTH.			
001680		MOVE TODAYS-DAY TO HDG-DAY.			
001690					
001700	0300	PROCESS-RECORDS.			
001710		PERFORM 0400-COMPUTE-GROSS-PAY.			
001720		PERFORM 0500-COMPUTE-FEDERAL-TAX.			
001730		PERFORM 0600-COMPUTE-NET-PAY.			
001740		PERFORM 0700-UPDATE-COMPANY-TOTALS.			

FIGURE 7.3 Continued

```

001750
001760 IF WS-LINE-COUNT > 3
001770     PERFORM 0800-WRITE-HEADING-LINE.
001780     PERFORM 0900-WRITE-DETAIL-LINE.
001790     ADD 1 TO WS-LINE-COUNT.
001800     READ EMPLOYEE-FILE
001810     AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
001820
001830 0800-WRITE-HEADING-LINE.
001840     ADD 1 TO WS-PAGE-COUNT.
001850     MOVE 1 TO WS-LINE-COUNT.
001860     MOVE WS-PAGE-COUNT TO HDG-PAGE-NUMBER.
001870     WRITE PRINT-LINE FROM HEADING-LINE-ONE
001880     AFTER ADVANCING PAGE.
001890     WRITE PRINT-LINE FROM HEADING-LINE-TWO
001900     AFTER ADVANCING 4 LINES.
001910     WRITE PRINT-LINE FROM DASHED-LINE
001920     AFTER ADVANCING 1 LINE.
001930
001940 0400-COMPUTE-GROSS-PAY.
001950     MULTIPLY EMP-REG-HOURS BY EMP-RATE GIVING IND-REGULAR-PAY.
001960     COMPUTE IND-OVERTIME-PAY
001970     = EMP-OVERTIME-HOURS * EMP-RATE * 1.5.
001980     ADD IND-REGULAR-PAY IND-OVERTIME-PAY GIVING IND-GROSS-PAY.
001990
002000 0500-COMPUTE-FEDERAL-TAX.
002010     COMPUTE IND-FEDERAL-TAX = .16 * IND-GROSS-PAY.
002020     IF IND-GROSS-PAY > 160
002030         COMPUTE IND-FEDERAL-TAX
002040         = IND-FEDERAL-TAX + .02 * (IND-GROSS-PAY - 160).
002050
002060     IF IND-GROSS-PAY > 200
002070         COMPUTE IND-FEDERAL-TAX
002080         = IND-FEDERAL-TAX + .02 * (IND-GROSS-PAY - 200).
002090
002100 0600-COMPUTE-NET-PAY.
002110     COMPUTE IND-NET-PAY = IND-GROSS-PAY - IND-FEDERAL-TAX.
002120
002130 0700-UPDATE-COMPANY-TOTALS.
002140     ADD IND-REGULAR-PAY TO CO-REGULAR-PAY.
002150     ADD IND-OVERTIME-PAY TO CO-OVERTIME-PAY.
002160     ADD IND-GROSS-PAY TO CO-GROSS-PAY.
002170     ADD IND-FEDERAL-TAX TO CO-FEDERAL-TAX.
002180     ADD IND-NET-PAY TO CO-NET-PAY.
002190
002200 0900-WRITE-DETAIL-LINE.
002210     MOVE SPACES TO DETAIL-LINE.
002220     MOVE EMP-LAST-NAME TO DET-LAST-NAME.
002230     MOVE EMP-RATE TO DET-RATE.
002240     MOVE EMP-REG-HOURS TO DET-REG-HOURS.
002250     MOVE EMP-OVERTIME-HOURS TO DET-OVERTIME-HOURS.
002260     MOVE IND-REGULAR-PAY TO DET-REGULAR-PAY.
002270     MOVE IND-OVERTIME-PAY TO DET-OVERTIME-PAY.
002280     MOVE IND-GROSS-PAY TO DET-GROSS-PAY.
002290     MOVE IND-FEDERAL-TAX TO DET-FEDERAL-TAX.
002300     MOVE IND-NET-PAY TO DET-NET-PAY.
002310
002320     WRITE PRINT-LINE FROM DETAIL-LINE
002330     AFTER ADVANCING 2 LINES.
002340
002350 1000-WRITE-COMPANY-TOTALS.
002360     WRITE PRINT-LINE FROM DASHED-LINE
002370     AFTER ADVANCING 1 LINE.
002380     MOVE CO-REGULAR-PAY TO TOTAL-REGULAR-PAY.
002390     MOVE CO-OVERTIME-PAY TO TOTAL-OVERTIME-PAY.
002400     MOVE CO-GROSS-PAY TO TOTAL-GROSS-PAY.
002410     MOVE CO-FEDERAL-TAX TO TOTAL-FEDERAL-TAX.
002420     MOVE CO-NET-PAY TO TOTAL-NET-PAY.
002430
002440     WRITE PRINT-LINE FROM TOTAL-LINE
002450     AFTER ADVANCING 2 LINES.

```

Test to invoke heading routine

WS-LINE-COUNT is incremented for each detail line

WS-LINE-COUNT is reset to 1 in heading routine

Paragraph names are sequenced and functional

New functions in expanded payroll

Existing modules have been expanded

Existing modules have been expanded

Blank lines are inserted before paragraph names

FIGURE 7.3 Continued

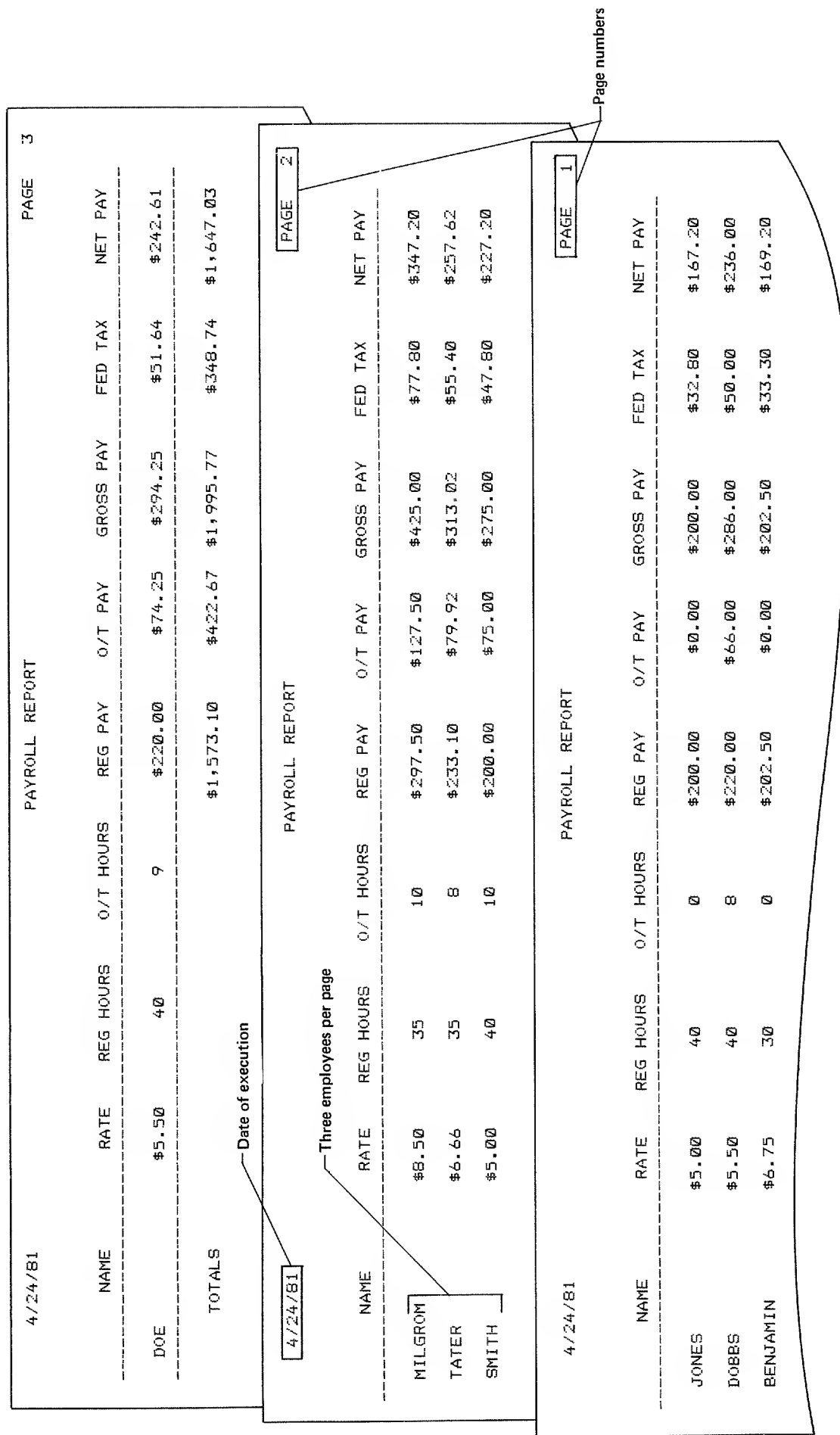


FIGURE 7.4 Expanded payroll report (produced by Figure 7.3)

Data names are fully descriptive. Related entries in Working-Storage are grouped together under a common 01 with a common prefix; e.g., IND-COMPUTATIONS (lines 560-610) and COMPANY-TOTALS (lines 630-680). Paragraph names are functional and sequenced.

Indentation is very important. It is used as a matter of course with level numbers in the Data Division (lines 320-400). It is also used in the Procedure Division where subservient clauses are indented, e.g., AT END under READ, UNTIL under PERFORM, and so on.

Output is shown in Figure 7.4. (The test data were identical to those in Figure 4.5.) Note well the page numbers and the presence of only three employees per page. WS-PAGE-COUNT and WS-LINE-COUNT are defined in lines 710 and 720. The value of WS-LINE-COUNT is tested *prior* to writing a detail line in line 1760. If it is greater than three, WRITE-HEADING-LINE is performed, which resets the line count and increments the page count. Note also that the line count is incremented after each detail line is written.

SUMMARY This important chapter discussed techniques for writing better programs. The reader was presented with a series of 12 coding guidelines which are thought to produce programs that are easy to read and maintain.

The chapter also defined the discipline of *structured programming*. This methodology has been in common use since the mid 1970s, and is applicable to small as well as large computers. The hierarchy chart was reviewed and seen to be a useful design aid.

The chapter ended with an expanded COBOL program of an earlier application, incorporating many of the techniques which were discussed.

TRUE/FALSE

1. COBOL requires that paragraph names be sequenced.
2. Blank lines are not permitted in a COBOL program.
3. A slash in column 7 causes the next line in a listing to begin on a new page.
4. 0100-READ-AND-COMPUTE is a good name for a COBOL paragraph.
5. Comments are indicated by an asterisk in column 7.
6. A program cannot have too many comments.
7. It is impossible to write a program without 77-level entries.
8. Two- and three-letter data names are desirable to reduce programmer coding time.
9. Indentation is a waste of time since it has no effect on the compiler.
10. COBOL requires all PICTURE clauses to begin in the same column.

PROJECTS

There are no new projects associated with this chapter. It is suggested, however, that the reader review the projects developed in earlier chapters with respect to programming style. Modify one or more of those projects to conform to the coding standards suggested in the present chapter.

8

**CONTROL
BREAKS**

OVERVIEW This chapter contains little that is new in the way of COBOL, but concentrates instead on the important data processing concept of *control breaks*. We progress from a simple example involving a single level control break to a more complex problem with two levels. Since the logic of the latter problem is somewhat involved, we emphasize the role of pseudocode as an aid in program development.

This chapter also extends the earlier discussion of editing to include all available characters. Editing, per se, is not required for processing control breaks. However, this unit presents a suitable place to introduce the material and to use it in subsequent programs. We begin, therefore, with a brief discussion of *editing*, which includes signed numbers, and then address the main thrust of the unit—control breaks.

EDITING Editing involves a change in data format. One may add commas, insert a dollar sign, suppress leading zeros, indicate negative values by a credit sign, etc. The purpose of all editing is to make reports easier to read.

The payroll problem of Chapter 4 introduced the concept of editing. Now we shall present a more complete discussion, beginning with Table 8.1, which contains the set of editing characters.

TABLE 8.1
EDITING CHARACTERS

<i>Symbol</i>	<i>Meaning</i>
.	Actual decimal point
Z	Zero suppress
*	Check protection
CR	Credit symbol
DB	Debit symbol
+	Plus sign
-	Minus sign
\$	Dollar sign
,	Comma
0	Zero
B	Blank
/	Slash

The blank (B) can be associated with any type of source field, i.e., alphabetic, numeric, or alphanumeric. The zero (0) is restricted to numeric or alphanumeric fields. All other symbols are for numeric source fields only. Realize, however, that any MOVE statement involving edited pictures is also affected by the rules discussed in Chapter 4 with regard to decimal alignment, truncation, etc. The use of various editing characters is best explained by direct example. Consider Table 8.2.

The concepts of source field, receiving field, and actual and assumed decimal point were covered in Chapter 4. The character Z will zero-suppress; i.e., it replaces leading zeros by blanks. The \$ causes a dollar sign to print. If several dollar signs are strung together [example (c) in Table 8.2], the effect is a floating dollar sign, i.e., a dollar sign prints immediately to the left of the first significant digit. The \$ can be used in conjunction with Z to cause a fixed dollar sign as in example (d) in Table 8.2. For obvious reasons, example (d) is not sound practice when “cutting” checks, and the asterisk is

TABLE 8.2
THE \$, Z, AND * EDIT CHARACTERS

Source Field		Receiving Field	
Picture	Value	Picture	Edited Result
(a) 9999V99	000123	9999.99	0001.23
(b) 9999V99	000123	ZZZZ.99	1.23
(c) 9999V99	000123	\$\$\$\$.99	\$1.23
(d) 9999V99	000123	\$ZZZZ.99	\$ 1.23
(e) 9999V99	000123	\$****.99	\$***1.23

used for check protection as shown in example (e). The asterisks appear in the edited result in lieu of leading zeros or spaces.

Signed Numbers

Frequently, the picture of a numeric source field is preceded by an S to indicate a signed field. The S is immaterial if only positive numbers can occur but absolutely essential any time a negative number results as the consequence of an arithmetic operation. *If the S is omitted, the result of the arithmetic operation will always assume a positive sign.* Consider:

```
05 FIELD-A    PIC S99    VALUE -20.
05 FIELD-B    PIC 99     VALUE 15.
05 FIELD-C    PIC S99    VALUE -20.
05 FIELD-D    PIC 99     VALUE 15.
```

```
ADD FIELD-B TO FIELD-A.
ADD FIELD-C TO FIELD-D.
```

Numerically, one expects the sum of -20 and +15 to be -5. If the result is stored in FIELD-A, there is no problem. However, if the sum is stored in FIELD-D (an unsigned field), it will assume a value of +5. Many programmers adopt the habit of always using signed fields to avoid any difficulty.

Table 8.3 illustrates the use of floating plus and minus signs. If a plus sign is used, the sign of the edited field will appear if the number is either positive, negative, or zero [examples (a), (b), and (c)]. However, if a minus sign is used, the sign appears only when the edited result is negative. Note also that the receiving field must be at least one character longer than the sending field to accommodate the sign; otherwise, a compiler warning message results.

Financial statements usually contain either the credit (CR.) or debit

TABLE 8.3
FLOATING + AND - CHARACTERS

Source Field		Receiving Field	
Picture	Value	Picture	Edited Result
(a) S9(4)	1234	++,+++	+1,234
(b) S9(4)	0123	++,+++	+123
(c) S9(4)	-1234	++,+++	-1,234
(d) S9(4)	1234	--,---	1,234
(e) S9(4)	0123	--,---	123
(f) S9(4)	-1234	--,---	-1,234

TABLE 8.4
CR AND DB SYMBOLS

Source Field		Receiving Field	
Picture	Value	Picture	Value
(a) S9(5)	98765	\$\$\$999CR	\$98,765
(b) S9(5)	-98765	\$\$\$999CR	\$98,765CR
(c) S9(5)	98765	\$\$\$999DB	\$98,765
(d) S9(5)	-98765	\$\$\$999DB	\$98,765DB

(DB) symbol to indicate a negative number. The use of these characters is illustrated in Table 8.4.

CR and DB appear only when the sending field is negative [examples (b) and (d)]. If the field is positive or zero, the symbols are replaced by blanks. The choice of CR or DB depends on the accounting system. COBOL treats both identically, i.e., CR and/or DB appear if and only if the sending field is negative.

CONTROL BREAKS

We now proceed to the main thrust of the chapter, control breaks. A *control break* is a change in a designated field. For example, if an incoming file has been arranged (i.e., sorted) so that all employees in the same location appear together, a control break occurs every time location changes. If the file is sorted by location, and department within location, it is possible to designate two control fields, department and location.

Let us assume that a file has been *sorted* on location, and department within location. Hence, all employees in department 100 in Atlanta precede the Atlanta employees in department 200, who precede those in department 300, etc. Next come the employees in department 100 in Boston, followed

	Name	Location	Department	Salary
Single control break	Adams	Atlanta	100	15,000
	Baker	Atlanta	100	18,000
	Davis	Atlanta	100	14,000
Single control break	Charles	Atlanta	200	19,000
	Lowell	Atlanta	200	17,500
	Smith	Atlanta	200	18,000
Double control break	Tyler	Atlanta	300	16,000
	Williams	Atlanta	300	20,000
Single control break	Abel	Boston	100	19,000
	Brewer	Boston	100	26,000
	Dixon	Boston	100	23,000
	Mason	Boston	100	19,000
	Murphy	Boston	100	16,500
Double control break	Cain	Boston	200	18,000
	Milgrom	Boston	200	17,000
	Atlas	Chicago	100	14,500

FIGURE 8.1 Control breaks

by department 200 in Boston, and so forth. A *single control break* occurs as we change departments within the same location; e.g., from department 100 in Atlanta to department 200 in Atlanta. A *double control break*, on location and department, arises when we go from department 300 in Atlanta to department 100 in Boston.

The situation is made clearer by Figure 8.1, which illustrates both single and double control breaks. As can be seen from Figure 8.1, the Atlanta employees appear together and precede the Boston employees, who in turn precede the Chicago employees, and so on. Moreover, the Atlanta employees have been grouped by department so that those in department 100 precede those in department 200, and so on.

A *single* control break occurs as we go from Davis to Charles, i.e., from department 100 to 200. A *double* control break results when we go from Williams (in department 300 in Atlanta) to Abel (in department 100 in Boston). Another double control break occurs when we go from Milgrom in Boston to Atlas in Chicago.

A constant data processing requirement is to process incoming files to compute control break totals. An example of such a report is shown in Figure 8.2, which totals salaries of all employees from the same department and the same location. Realize that Figure 8.2 shows only one location, i.e., Atlanta, and that there would be a similar page for each location in the complete report.

<i>Name</i>	<i>Location</i>	<i>Department</i>	<i>Salary</i>	
Adams	Atlanta	100	15,000	
Baker	Atlanta	100	18,000	
Davis	Atlanta	100	14,000	
			Total for department 100 =	47,000
Charles	Atlanta	200	19,000	
Lowell	Atlanta	200	17,500	
Smith	Atlanta	200	18,000	
			Total for department 200 =	54,500
Tyler	Atlanta	300	16,000	
Williams	Atlanta	300	20,000	
			Total for department 300 =	36,000
			Total for Atlanta =	137,500

FIGURE 8.2 Control break totals

ONE-LEVEL CONTROL BREAKS

We now develop a COBOL program to compute totals for a single control break. (The program will be expanded in the next section to cover two levels. The COBOL per se is not difficult. However, the logic required for the latter problem is as involved as anything the reader is apt to encounter.)

The problem is a marketing application. Input is a sales file, (Figure 8.3a) with each record containing information about a single transaction for a particular salesman. There can be several records for the same salesman.

The problem is to compute the total sales for each salesman, as well as the company total. To complicate matters, some of the transactions reflect returns rather than purchases, and are to be deducted from the total. Each

JONES	111111050000	SATLANTA
JONES	222222666666	SATLANTA
SMITH	100000300000	SATLANTA
SMITH	400000070000	RATLANTA
STOCKWELL	878787123456	RBOSTON
FORD	987654200000	SBOSTON
FORD	444333100000	SBOSTON
FORD	555666200000	SBOSTON

SALES ACTIVITY REPORT		
SALESMAN: FORD		LOCATION: BOSTON
ACCOUNT #:	RETURNS	SALES
987654		\$2,000.00
444333		\$1,000.00
555666		\$2,000.00
*** SALESMAN TOTAL = \$ 5,000.00		
*** COMPANY TOTAL = \$ 10,532.10		

SALES ACTIVITY REPORT		
SALESMAN: STOCKWELL		LOCATION: BOSTON
ACCOUNT #:	RETURNS	SALES
878787	\$1,234.56	
*** SALESMAN TOTAL = \$ 1,234.56CR		

SALES ACTIVITY REPORT		
SALESMAN: SMITH		LOCATION: ATLANTA
ACCOUNT #:	RETURNS	SALES
100000		\$ 300.00
400000	\$ 700.00	
*** SALESMAN TOTAL = \$ 400.00CR		

CR indicates a negative number;
i.e., returns greater than sales

SALES ACTIVITY REPORT		
SALESMAN: JONES		LOCATION: ATLANTA
ACCOUNT #:	RETURNS	SALES
111111		\$ 500.00
222222		\$6,666.66
*** SALESMAN TOTAL = \$ 7,166.66		

FIGURE 8.3 (a) Test data. (b) One-level control breaks (output)

transaction, therefore, contains a code, S or R, indicating a sale or return, respectively.

Figure 8.3 shows hypothetical data and the desired report. Note well that the incoming data have been sorted by salesman so that all transactions for the same individual appear together. The output for each salesman appears on a page by itself. Observe also that both transactions for Jones are sales, and hence his total represents the sum of the transactions. Smith, however, had one sale and one return. Moreover, the amount of the return was greater than the amount of sale, and thus his total is negative as indicated by the CR editing characters.

Pseudocode for the program is expressed in Figure 8.4. The program begins with typical "housekeeping" functions such as opening files and an initial read. Next, it processes transactions until there are no more data, after which the final total is printed and the program terminates.

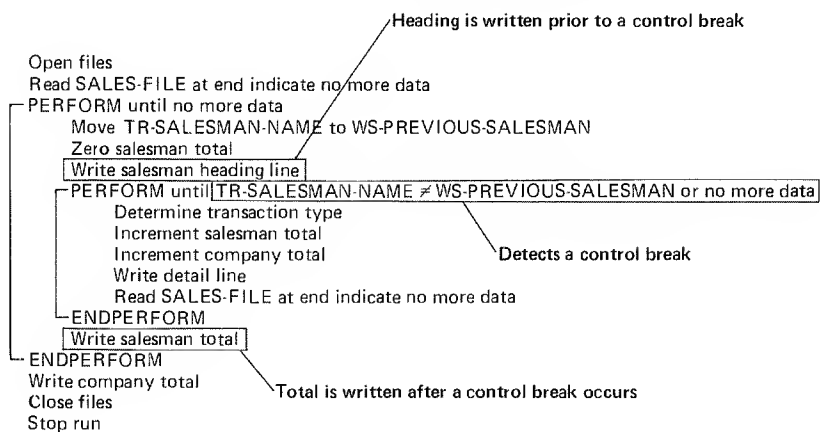


FIGURE 8.4 Pseudocode for single control break

Note well that the pseudocode (and eventual program) must contain a mechanism for detecting a control break. Accordingly, after a transaction is read, TR-SALESMAN-NAME is moved to a second data name, WS-PREVIOUS-SALESMAN. The pseudocode then processes transactions until a control break occurs; i.e., TR-SALESMAN-NAME is *not* equal to WS-PREVIOUS-SALESMAN. The total for the current salesman is written, after which WS-PREVIOUS-SALESMAN is reset and the process continues.

The completed program is shown in Figure 8.5. Note that TRANSACTION-WORK-AREA defines two 88-level entries under TR-CODE (lines 540-560). Hence, in the IF statement of line 1320, the entry IF SALE is equivalent to IF TR-CODE = "S". Condition names (88-level entries) were introduced in Chapter 6 as a means of enhancing clarity in the Procedure Division.

Observe also that TR-AMOUNT is either added or subtracted depending on a sale or return. Consequently, both THIS-SALESMAN-TOTAL and COMPANY-TOTAL are defined as signed numbers (lines 460 and 470) to accommodate potential negative numbers. Realize also that both THIS-SALESMAN-TOTAL and COMPANY-TOTAL are used for calculations only; the printed output is contained in PRT-SALESMAN-TOTAL and PRT-COMPANY-TOTAL, lines 960 and 1030, respectively. The latter are edited fields containing the characters CR, which print if and only if the sending field is negative.

The Procedure Division is fairly straightforward. The files are opened and an initial read is executed. The paragraph, 020-PROCESS-ALL-SALESMEN,

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID. ONELEVEL.
000120 AUTHOR. R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER. TRS-80.
000170 OBJECT-COMPUTER. TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210     SELECT SALES-FILE
000220         ASSIGN TO INPUT "ONELEVEL/DAT".
000230     SELECT PRINT-FILE
000240         ASSIGN TO PRINT "ONELEVEL/TXT".
000250
000260 DATA DIVISION.
000270 FILE SECTION.
000280 FD SALES-FILE
000290     LABEL RECORDS ARE STANDARD
000300     RECORD CONTAINS 47 CHARACTERS
000310     DATA RECORD IS SALES-RECORD.
000320 01 SALES-RECORD PIC X(47).
000330
000340 FD PRINT-FILE
000350     LABEL RECORDS ARE STANDARD
000360     RECORD CONTAINS 132 CHARACTERS
000370     DATA RECORD IS PRINT-LINE.
000380 01 PRINT-LINE PIC X(132).
000390
000400 WORKING-STORAGE SECTION.
000410 01 PROGRAM-SWITCHES.
000420     05 WS-DATA-REMAINS-SWITCH PIC X(3) VALUE "YES".
000430     05 WS-PREVIOUS-SALESMAN PIC X(15) VALUE SPACES.
000440
000450 01 CONTROL-BREAK-TOTALS.
000460     05 THIS-SALESMAN-TOTAL PIC S9(6)V99 VALUE ZEROS.
000470     05 COMPANY-TOTAL PIC S9(6)V99 VALUE ZEROS.
000480
000490 01 TRANSACTION-WORK-AREA.
000500     05 FILLER PIC X(4).
000510     05 TR-SALESMAN-NAME PIC X(15).
000520     05 TR-ACCOUNT-NUMBER PIC 9(6).
000530     05 TR-AMOUNT PIC 9(4)V99.
000540     05 TR-CODE PIC X.
000550         88 RETURNS VALUE "R".
000560         88 SALE VALUE "S".
000570     05 TR-LOCATION PIC X(15).
000580
000590 01 HDG-LINE-ONE.
000600     05 FILLER PIC X(22) VALUE SPACES.
000610     05 FILLER PIC X(21)
000620         VALUE "SALES ACTIVITY REPORT".
000630     05 FILLER PIC X(89) VALUE SPACES.
000640
000650 01 HDG-LINE-TWO.
000660     05 FILLER PIC X(5) VALUE SPACES.
000670     05 FILLER PIC X(10) VALUE "SALESMAN: ".
000680     05 HDG-NAME PIC X(15).
000690     05 FILLER PIC X(25) VALUE SPACES.
000700     05 FILLER PIC X(10) VALUE "LOCATION: ".
000710     05 HDG-LOCATION PIC X(13).
000720     05 FILLER PIC X(54) VALUE SPACES.
000730
000740 01 HDG-LINE-THREE.
000750     05 FILLER PIC X(10) VALUE SPACES.
000760     05 FILLER PIC X(11) VALUE "ACCOUNT #: ".
000770     05 FILLER PIC X(15) VALUE SPACES.
000780     05 FILLER PIC X(7) VALUE "RETURNS".
000790     05 FILLER PIC X(15) VALUE SPACES.

```

— Detects end of input file

— Tests for control break

— "S" indicates a signed number

— 88 level entry

FIGURE 8.5 One-level control break program

```

000800      05 FILLER                      PIC X(5)      VALUE "SALES".
000810      05 FILLER                      PIC X(69)     VALUE SPACES.
000820
000830 01  DETAIL-LINE.
000840      05 FILLER                      PIC X(14)     VALUE SPACES.
000850      05 DET-ACCOUNT-NUMBER          PIC 9(6).
000860      05 FILLER                      PIC X(14)     VALUE SPACES.
000870      05 DET-RETURNS                 PIC $Z,ZZ9.99.
000880      05 FILLER                      PIC X(11)     VALUE SPACES.
000890      05 DET-SALES                   PIC $Z,ZZ9.99.
000900      05 FILLER                      PIC X(69)     VALUE SPACES.
000910
000920 01  SALESMAN-TOTAL-LINE.
000930      05 FILLER                      PIC X(25)     VALUE SPACES.
000940      05 FILLER                      PIC X(21)
000950      VALUE "*** SALESMAN TOTAL = ".
000960      05 PRT-SALESMAN-TOTAL          PIC $Z(3),ZZ9.99CR.
000970      05 FILLER                      PIC X(73)     VALUE SPACES.
000980
000990 01  COMPANY-TOTAL-LINE.
001000      05 FILLER                      PIC X(26)     VALUE SPACES.
001010      05 FILLER                      PIC X(20)
001020      VALUE "*** COMPANY TOTAL = ".
001030      05 PRT-COMPANY-TOTAL           PIC $Z(3),ZZ9.99CR.
001040      05 FILLER                      PIC X(73)     VALUE SPACES.
001050
001060 PROCEDURE DIVISION.
001070 010-CALCULATE-CONTROL-BREAKS.
001080     OPEN INPUT SALES-FILE
001090     OUTPUT PRINT-FILE.
001100     READ SALES-FILE INTO TRANSACTION-WORK-AREA
001110     AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
001120     PERFORM 020-PROCESS-ALL-SALESMEN
001130     UNTIL WS-DATA-REMAINS-SWITCH = "NO".
001140     PERFORM 080-WRITE-COMPANY-TOTAL.
001150     CLOSE SALES-FILE
001160     PRINT-FILE.
001170     STOP RUN.
001180
001190 020-PROCESS-ALL-SALESMEN.
001200     MOVE TR-SALESMAN-NAME TO WS-PREVIOUS-SALESMAN.
001210     MOVE ZEROS TO THIS-SALESMAN-TOTAL.
001220     PERFORM 060-WRITE-SALESMAN-HEADING.
001230     PERFORM 030-PROCESS-ALL-TRANSACTIONS
001240     UNTIL TR-SALESMAN-NAME NOT EQUAL WS-PREVIOUS-SALESMAN
001250     OR WS-DATA-REMAINS-SWITCH = "NO".
001260     PERFORM 070-WRITE-SALESMAN-TOTAL.
001270
001280 030-PROCESS-ALL-TRANSACTIONS.
001290     MOVE SPACES TO DETAIL-LINE.
001300     MOVE TR-ACCOUNT-NUMBER TO DET-ACCOUNT-NUMBER.
001310
001320     IF SALE
001330         MOVE TR-AMOUNT TO DET-SALES
001340         ADD TR-AMOUNT TO THIS-SALESMAN-TOTAL
001350         ADD TR-AMOUNT TO COMPANY-TOTAL
001360     ELSE
001370         IF RETURNS
001380             MOVE TR-AMOUNT TO DET-RETURNS
001390             SUBTRACT TR-AMOUNT FROM THIS-SALESMAN-TOTAL
001400             SUBTRACT TR-AMOUNT FROM COMPANY-TOTAL.
001410
001420     WRITE PRINT-LINE FROM DETAIL-LINE
001430     AFTER ADVANCING 2 LINES.
001440     READ SALES-FILE INTO TRANSACTION-WORK-AREA
001450     AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
001460 060-WRITE-SALESMAN-HEADING.
001470     WRITE PRINT-LINE FROM HDG-LINE-ONE
001480     AFTER ADVANCING PAGE.

```

Edited pictures

Prints only if the sending field is negative (see lines 460 and 1570)

Initializes each salesman's total

Tests for control break

Writes one salesman's total

Totals are incremented or decremented, depending on TR-CODE (see lines 540-560)

FIGURE 8.5 Continued


```

001490      MOVE TR-SALESMAN-NAME TO HDG-NAME.
001500      MOVE TR-LOCATION TO HDG-LOCATION.
001510      WRITE PRINT-LINE FROM HDG-LINE-TWO
001520          AFTER ADVANCING 2 LINES.
001530      WRITE PRINT-LINE FROM HDG-LINE-THREE
001540          AFTER ADVANCING 3 LINES.
001550
001560 070-WRITE-SALESMAN-TOTAL.
001570      MOVE THIS-SALESMAN-TOTAL TO PRT-SALESMAN-TOTAL.
001580      WRITE PRINT-LINE FROM SALESMAN-TOTAL-LINE
001590          AFTER ADVANCING 2 LINES.
001600
001610 080-WRITE-COMPANY-TOTAL.
001620      MOVE COMPANY-TOTAL TO PRT-COMPANY-TOTAL.
001630      WRITE PRINT-LINE FROM COMPANY-TOTAL-LINE
001640          AFTER ADVANCING 5 LINES.

```

FIGURE 8.5 *Continued*

is executed as long as data remain, after which the company total is written and the program terminates.

The PROCESS-ALL-SALESMEN paragraph is the key to the program. The current salesman, TR-SALESMAN-NAME is moved to WS-PREVIOUS-SALESMAN as a means for detecting subsequent control breaks. The salesman's total is initialized to zero and a heading is written. A lower level paragraph is performed to process individual transactions until TR-SALESMAN-NAME is unequal to WS-PREVIOUS-SALESMAN; i.e., the control break occurs. The salesman's total is written and the routine PROCESS-ALL-SALESMEN is re-executed until the input file is exhausted.

TWO-LEVEL CONTROL BREAKS

The previous example is extended to a second level, in which totals are required for individual salesmen (as before), and for each city as well. Figure 8.6 contains both test data and the desired report. The test data for this example (Figure 8.6a) are the same as in the previous example (Figure 8.3a).

Observe carefully the order of the incoming records. All salesmen in the same city appear together. Moreover, the multiple transactions for a given salesman are also consecutive. In other words, the incoming file has been sorted by city, and by salesman within city.

Development of the subsequent program is facilitated by first considering the hierarchy chart of Figure 8.7. Recall that a hierarchy chart illustrates *function, not procedure*; i.e., it shows what has to be done but not when. Remember also, that each box in the hierarchy chart will become a paragraph in the completed program.

The "boss" of the program is CALCULATE-CONTROL-BREAKS. It in turn calls two modules, PROCESS-ALL-LOCATIONS and WRITE-COMPANY-TOTAL. Each new location requires a heading, processing of all its salesmen, and a total; accomplished by WRITE-LOCATION-HEADING, PROCESS-ALL-SALESMEN, and WRITE-LOCATION-TOTAL, respectively. Each salesman in turn requires his own heading, processing of all his transactions, and a total; accomplished by the three lowest level modules on the hierarchy chart.

The pseudocode of Figure 8.8 illustrates procedure rather than function; i.e., it shows when and how various processing is accomplished. It is similar to the pseudocode of Figure 8.4, except that it has been expanded to accommodate two control breaks. Pseudocode and/or a hierarchy chart are useful as design tools in developing the program of Figure 8.9.

JONES	111111050000SATLANTA
JONES	222222666666SATLANTA
SMITH	100000030000SATLANTA
SMITH	400000070000RATLANTA
STOCKWELL	878787123456RBOSTON
FORD	987654200000SBOSTON
FORD	444333100000SBOSTON
FORD	555666200000SBOSTON

SALES ACTIVITY REPORT - BOSTON

SALESMAN: STOCKWELL

ACCOUNT #:	RETURNS	SALES
878787	\$1,234.56	

*** SALESMAN TOTAL = \$ 1,234.56CR

SALESMAN: FORD

ACCOUNT #:	RETURNS	SALES
987654		\$2,000.00
444333		\$1,000.00
555666		\$2,000.00

*** SALESMAN TOTAL = \$ 5,000.00

*** LOCATION TOTAL = \$ 3,765.44

*** COMPANY TOTAL = \$ 10,532.10

SALES ACTIVITY REPORT - ATLANTA

SALESMAN: JONES

ACCOUNT #:	RETURNS	SALES
111111		\$ 500.00
222222		\$6,666.66

*** SALESMAN TOTAL = \$ 7,166.66

SALESMAN: SMITH

ACCOUNT #:	RETURNS	SALES
100000		\$ 300.00
400000	\$ 700.00	

*** SALESMAN TOTAL = \$ 400.00CR

*** LOCATION TOTAL = \$ 6,766.66

FIGURE 8.6 Test data and report for two-level control break. (a) Test data. (b) Report

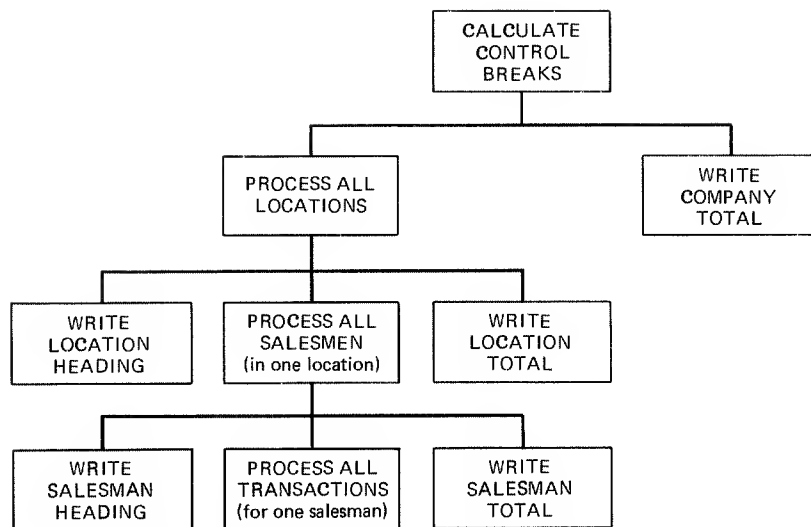


FIGURE 8.7 Hierarchy chart for two-level control break program

The two-level program uses the same input file as the earlier one-level version (lines 210–220). Note, however, the additional data names, WS-PREVIOUS-LOCATION and THIS-LOCATION-TOTAL (lines 440 and 480) which are necessary for the second control break. The Procedure Division at first appears somewhat intimidating. The hierarchy chart, however, clearly shows the relationship of the paragraphs, and thereby makes the program easier to follow. Thus, we see that the hierarchy chart serves *two* purposes. It is both a *design* and *documentation* aid. It is useful *before* the program is written to indicate what functions the programmer must include (the design function). It is also helpful after the fact to explain how a program works (documentation).

Little more needs to be said about the program itself. The reader should be able to follow the program because of the associated documentation, and

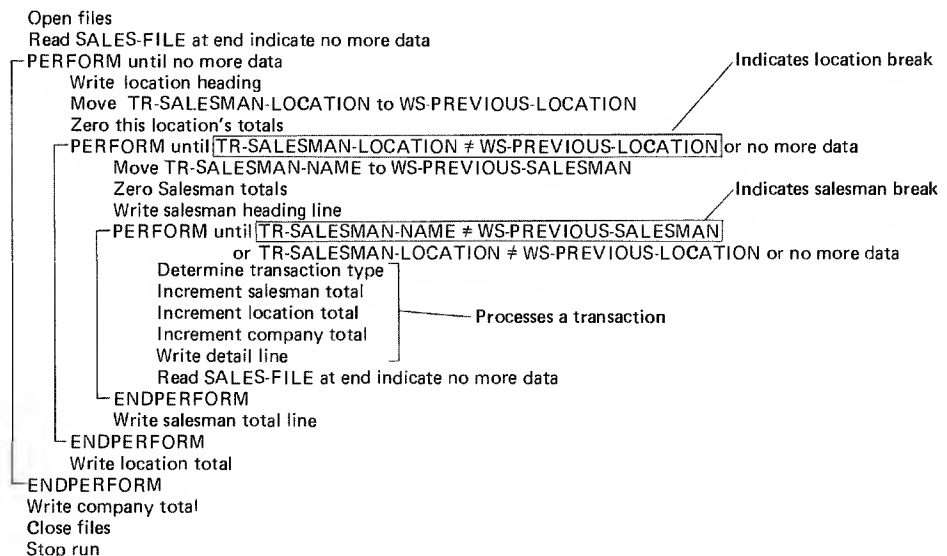


FIGURE 8.8 Pseudocode for double control break program

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID. TWOLEVEL.
000120 AUTHOR. R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER. TRS-80.
000170 OBJECT-COMPUTER. TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210     SELECT SALES-FILE
000220         ASSIGN TO INPUT "ONELEVEL/DAT".
000230     SELECT PRINT-FILE
000240         ASSIGN TO PRINT "TWOLEVEL/TXT".
000250
000260 DATA DIVISION.
000270 FILE SECTION.
000280 FD  SALES-FILE
000290     LABEL RECORDS ARE STANDARD
000300     RECORD CONTAINS 47 CHARACTERS
000310     DATA RECORD IS SALES-RECORD.
000320 01  SALES-RECORD                PIC X(47).
000330
000340 FD  PRINT-FILE
000350     LABEL RECORDS ARE STANDARD
000360     RECORD CONTAINS 132 CHARACTERS
000370     DATA RECORD IS PRINT-LINE.
000380 01  PRINT-LINE                  PIC X(132).
000390
000400 WORKING-STORAGE SECTION.
000410 01  PROGRAM-SWITCHES.
000420     05  WS-DATA-REMAINS-SWITCH    PIC X(3)    VALUE "YES".
000430     05  WS-PREVIOUS-SALESMAN      PIC X(15)   VALUE SPACES.
000440     05  WS-PREVIOUS-LOCATION        PIC X(15)   VALUE SPACES.
000450
000460 01  CONTROL-BREAK-TOTALS.
000470     05  THIS-SALESMAN-TOTAL        PIC S9(6)V99  VALUE ZEROS.
000480     05  THIS-LOCATION-TOTAL          PIC S9(6)V99  VALUE ZEROS.
000490     05  COMPANY-TOTAL              PIC S9(6)V99  VALUE ZEROS.
000500
000510 01  TRANSACTION-WORK-AREA.
000520     05  FILLER                     PIC X(4).
000530     05  TR-SALESMAN-NAME            PIC X(15).
000540     05  TR-ACCOUNT-NUMBER           PIC 9(6).
000550     05  TR-AMOUNT                   PIC 9(4)V99.
000560     05  TR-CODE                     PIC X.
000570     05  88 RETURNS                  VALUE "R".
000580     05  88 SALE                      VALUE "S".
000590     05  TR-SALESMAN-LOCATION          PIC X(15).
000600
000610 01  HDG-LINE-ONE.
000620     05  FILLER                     PIC X(15)    VALUE SPACES.
000630     05  FILLER                     PIC X(24)
000640         VALUE "SALES ACTIVITY REPORT - ".
000650     05  HDG-LOCATION                 PIC X(15)    VALUE SPACES.
000660     05  FILLER                     PIC X(78)    VALUE SPACES.
000670
000680 01  HDG-LINE-TWO.
000690     05  FILLER                     PIC X(5)     VALUE SPACES.
000700     05  FILLER                     PIC X(10)    VALUE "SALESMAN: ".
000710     05  HDG-NAME                   PIC X(15).
000720     05  FILLER                     PIC X(25)    VALUE SPACES.
000730     05  FILLER                     PIC X(77)    VALUE SPACES.
000740
000750 01  HDG-LINE-THREE.
000760     05  FILLER                     PIC X(10)    VALUE SPACES.
000770     05  FILLER                     PIC X(11)    VALUE "ACCOUNT #: ".
000780     05  FILLER                     PIC X(15)    VALUE SPACES.
000790     05  FILLER                     PIC X(7)     VALUE "RETURNS".

```

Uses the same file as one level program

Location switch has been added for second control break

Signed Picture clauses for potentially negative numbers

Input records contain location for second control break

FIGURE 8.9 Two-level control break program

000800	05	FILLER	PIC X(15)	VALUE SPACES.	
000810	05	FILLER	PIC X(5)	VALUE "SALES".	
000820	05	FILLER	PIC X(69)	VALUE SPACES.	
000830					
000840	01	DETAIL-LINE.			
000850	05	FILLER	PIC X(14)	VALUE SPACES.	
000860	05	DET-ACCOUNT-NUMBER	PIC 9(6).		
000870	05	FILLER	PIC X(14)	VALUE SPACES.	
000880	05	DET-RETURNS	PIC \$Z,ZZ9.99.		
000890	05	FILLER	PIC X(11)	VALUE SPACES.	
000900	05	DET-SALES	PIC \$Z,ZZ9.99.		
000910	05	FILLER	PIC X(69)	VALUE SPACES.	
000920					
000930	01	SALESMAN-TOTAL-LINE.			
000940	05	FILLER	PIC X(25)	VALUE SPACES.	
000950	05	FILLER	PIC X(21)		
000960		VALUE "*** SALESMAN TOTAL = ".			
000970	05	PRT-SALESMAN-TOTAL	PIC \$Z(3),ZZ9.99	CR.	
000980	05	FILLER	PIC X(73)	VALUE SPACES.	
000990					
001000	01	LOCATION-TOTAL-LINE.			
001010	05	FILLER	PIC X(25)	VALUE SPACES.	
001020	05	FILLER	PIC X(21)		
001030		VALUE "*** LOCATION TOTAL = ".			
001040	05	PRT-LOCATION-TOTAL	PIC \$Z(3),ZZ9.99	CR.	
001050	05	FILLER	PIC X(73)	VALUE SPACES.	
001060					
001070	01	COMPANY-TOTAL-LINE.			
001080	05	FILLER	PIC X(26)	VALUE SPACES.	
001090	05	FILLER	PIC X(20)		
001100		VALUE "*** COMPANY TOTAL = ".			
001110	05	PRT-COMPANY-TOTAL	PIC \$Z(3),ZZ9.99	CR.	
001120	05	FILLER	PIC X(73)	VALUE SPACES.	
001130					
001140		PROCEDURE DIVISION.			
001150	010	CALCULATE-CONTROL-BREAKS.			
001160		OPEN INPUT SALES-FILE			
001170		OUTPUT PRINT-FILE.			
001180		READ SALES-FILE INTO TRANSACTION-WORK-AREA			
001190		AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.			
001200		PERFORM 015-PROCESS-ALL-LOCATIONS			
001210		UNTIL WS-DATA-REMAINS-SWITCH = "NO".			
001220		PERFORM 080-WRITE-COMPANY-TOTAL.			
001230		CLOSE SALES-FILE			
001240		PRINT-FILE.			
001250		STOP RUN.			
001260					
001270	015	PROCESS-ALL-LOCATIONS.			
001280		PERFORM 065-WRITE-LOCATION-HEADING.			
001290		MOVE TR-SALESMAN-LOCATION TO WS-PREVIOUS-LOCATION.			
001300		MOVE ZEROS TO THIS-LOCATION-TOTAL.			
001310		PERFORM 020-PROCESS-ALL-SALESMEN			
001320		UNTIL TR-SALESMAN-LOCATION NOT EQUAL WS-PREVIOUS-LOCATION			
001321		OR WS-DATA-REMAINS-SWITCH = "NO".			
001330		PERFORM 075-WRITE-LOCATION-TOTAL.			
001340					
001350	020	PROCESS-ALL-SALESMEN.			
001360		MOVE TR-SALESMAN-NAME TO WS-PREVIOUS-SALESMAN.			
001370		MOVE ZEROS TO THIS-SALESMAN-TOTAL.			
001380		PERFORM 060-WRITE-SALESMAN-HEADING.			
001390		PERFORM 030-PROCESS-ALL-TRANSACTIONS			
001400		UNTIL TR-SALESMAN-NAME NOT EQUAL WS-PREVIOUS-SALESMAN			
001405		OR TR-SALESMAN-LOCATION NOT EQUAL WS-PREVIOUS-LOCATION			
001410		OR WS-DATA-REMAINS-SWITCH = "NO".			
001420		PERFORM 070-WRITE-SALESMAN-TOTAL.			
001430					
001440	030	PROCESS-ALL-TRANSACTIONS.			
001450		MOVE SPACES TO DETAIL-LINE.			
001460		MOVE TR-ACCOUNT-NUMBER TO DET-ACCOUNT-NUMBER.			

Appears only if sending
field is negative

Control Break on Location

Control Break on Salesman

FIGURE 8.9 Continued

```

001480 IF SALE
001490     MOVE TR-AMOUNT TO DET-SALES
001500     ADD TR-AMOUNT TO THIS-SALESMAN-TOTAL
001510     ADD TR-AMOUNT TO THIS-LOCATION-TOTAL
001520     ADD TR-AMOUNT TO COMPANY-TOTAL
001530 ELSE
001531     IF RETURNS
001540         MOVE TR-AMOUNT TO DET-RETURNS
001550         SUBTRACT TR-AMOUNT FROM THIS-SALESMAN-TOTAL
001560         SUBTRACT TR-AMOUNT FROM THIS-LOCATION-TOTAL
001570         SUBTRACT TR-AMOUNT FROM COMPANY-TOTAL.
001580
001590 WRITE PRINT-LINE FROM DETAIL-LINE
001600     AFTER ADVANCING 1 LINE.
001610 READ SALES-FILE INTO TRANSACTION-WORK-AREA
001620     AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
001630
001640 060-WRITE-SALESMAN-HEADING.
001650     MOVE TR-SALESMAN-NAME TO HDG-NAME.
001660     WRITE PRINT-LINE FROM HDG-LINE-TWO
001670         AFTER ADVANCING 2 LINES.
001680     WRITE PRINT-LINE FROM HDG-LINE-THREE
001690         AFTER ADVANCING 3 LINES.
001700
001710 065-WRITE-LOCATION-HEADING.
001720     MOVE TR-SALESMAN-LOCATION TO HDG-LOCATION.
001730     WRITE PRINT-LINE FROM HDG-LINE-ONE
001740         AFTER ADVANCING PAGE.
001750
001760 070-WRITE-SALESMAN-TOTAL.
001770     MOVE THIS-SALESMAN-TOTAL TO PRT-SALESMAN-TOTAL.
001780     WRITE PRINT-LINE FROM SALESMAN-TOTAL-LINE
001790         AFTER ADVANCING 2 LINES.
001800
001810 075-WRITE-LOCATION-TOTAL.
001820     MOVE THIS-LOCATION-TOTAL TO PRT-LOCATION-TOTAL.
001830     WRITE PRINT-LINE FROM LOCATION-TOTAL-LINE
001840         AFTER ADVANCING 2 LINES.
001850
001860 080-WRITE-COMPANY-TOTAL.
001870     MOVE COMPANY-TOTAL TO PRT-COMPANY-TOTAL.
001880     WRITE PRINT-LINE FROM COMPANY-TOTAL-LINE
001890         AFTER ADVANCING 5 LINES.

```

Totals are incremented or decremented according to 88 level entry

Totals are printed at different times depending on control break

FIGURE 8.9 Continued

because of the attention to programming style. The author takes considerable effort in adhering to the coding guidelines suggested in Chapter 7; e.g., indentation, meaningful data names, and so on.

SUMMARY This chapter focused on control breaks, one of the most frequent data processing applications. We began by defining a control break as a change in a designated field. We developed two programs, for one and two level breaks, respectively. Nothing new in the way of COBOL is required to process control breaks. However, the logic itself was certainly nontrivial. Consequently, emphasis was placed on the pseudocode and hierarchy charts associated with each of the example programs. The importance of programming style, i.e., meaningful data names, indentation, and so on was stressed as a way of making programs easier to read.

The chapter also completed an earlier discussion on editing. Numerous examples were presented to illustrate the various editing characters. Signed numbers were found to be especially important, with attention directed to the CR and DB editing characters, as well as floating plus and minus signs.

TRUE/FALSE

1. The characters CR and DB represent plus and minus signs, respectively.
2. The characters \$ and * may not appear in the same picture clause.
3. An unsigned numeric field may hold negative numbers.
4. Arithmetic can be performed directly on a field with a dollar sign or comma in its picture.
5. The picture \$\$\$9.99 is preferable to a picture of \$***.*9.
6. Input records to a program processing control breaks need not be in any special order.
7. Control break processing must be restricted to a single level.
8. Counters in a COBOL program are automatically initialized to zero by the COBOL compiler.
9. Control breaks are an infrequent application.
10. Indentation and meaningful data names are a waste of time in COBOL.

EXERCISES

1. Show the edited results for each entry:

Source Field		Receiving Field	
Picture	Value	Picture	Edited Result
(a) S9(4)V99	-045600	\$\$\$\$.99CR	
(b) S9(4)V99	045600	\$\$,\$\$\$99DB	
(c) S9(4)	4567	\$\$,\$\$\$00	
(d) S9(6)	122577	99B99B99	
(e) S9(6)	123456	++++,+++	
(f) S9(6)	-123456	++++,+++	
(g) S9(6)	123456	---,---	
(h) S9(6)	-123456	---,---	
(i) 9(6)V99	00567890	\$\$\$\$,\$\$\$99	
(j) 9(6)V99	00567890	\$\$\$Z,\$\$\$99	
(k) 9(6)V99	00567890	\$***,***.99	

2. Consider the following code:

```

01 AMOUNT-REMAINING      PIC 9(3) VALUE 100.
01 WS-INPUT-AREA.
   05 QUANTITY-SHIPED    PIC 99.
   05 REST-OF-A-RECORD   PIC X(50).
.
.
READ TRANSACTION-FILE INTO WS-INPUT-AREA
  AT END MOVE "YES" TO EOF-SWITCH.
PERFORM PROCESS-TRANSACTIONS
  UNTIL EOF-SWITCH = "YES"
    OR AMOUNT-REMAINING < 0.
.
.
PROCESS-TRANSACTIONS.
  SUBTRACT QUANTITY-SHIPED FROM AMOUNT-REMAINING.
  READ TRANSACTION-FILE INTO WS-INPUT-AREA
    AT END MOVE "YES" TO EOF-SWITCH.

```

- (a) Why will AMOUNT-REMAINING *never* be less than zero?
 (b) What will be the final value of AMOUNT-REMAINING, given successive values of 30, 50, 25, and 15 for QUANTITY-SHIPPED?
3. *Debugging:* Figure 8.10 contains *invalid* output produced by the *incorrect* program of Figure 8.11. The data and intended results were shown in Figures 8.6a and 8.6b, respectively.
- Find and correct the errors in the program of Figure 8.11 so that it works as intended. (The reader may observe that Figure 8.11 is a modified version of Figure 8.9, which was developed correctly in the chapter.)

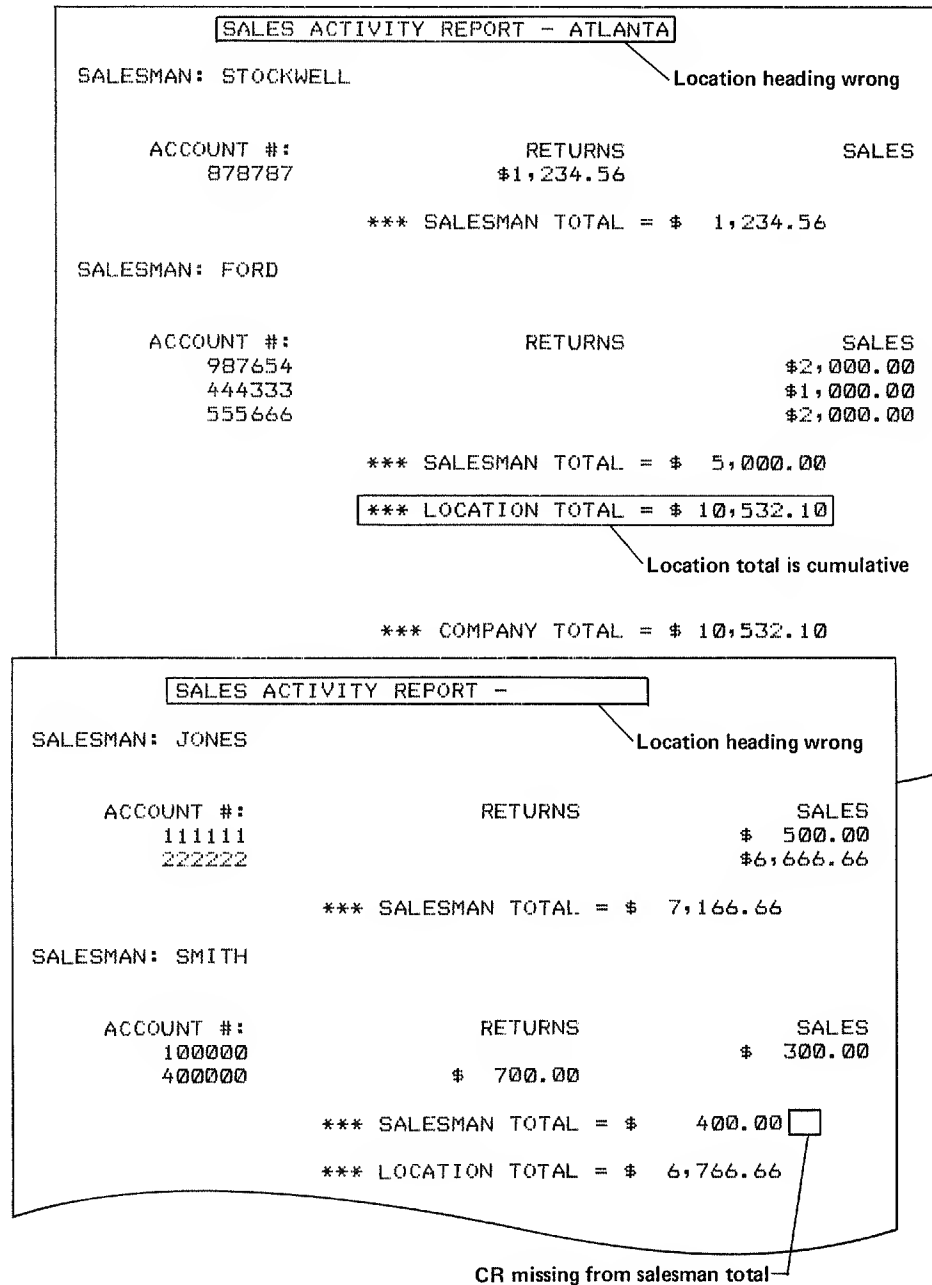


FIGURE 8.10 Invalid output (produced by Figure 8.11)


```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID. ETWOLEVE.
000120 AUTHOR. R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER. TRS-80.
000170 OBJECT-COMPUTER. TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210     SELECT SALES-FILE
000220         ASSIGN TO INPUT "ONELEVEL/DAT".
000230     SELECT PRINT-FILE
000240         ASSIGN TO PRINT "ETWOLEVE/TXT".
000250
000260 DATA DIVISION.
000270 FILE SECTION.
000280 FD SALES-FILE
000290     LABEL RECORDS ARE STANDARD
000300     RECORD CONTAINS 47 CHARACTERS
000310     DATA RECORD IS SALES-RECORD.
000320 01 SALES-RECORD                PIC X(47).
000330
000340 FD PRINT-FILE
000350     LABEL RECORDS ARE STANDARD
000360     RECORD CONTAINS 132 CHARACTERS
000370     DATA RECORD IS PRINT-LINE.
000380 01 PRINT-LINE                  PIC X(132).
000390
000400 WORKING-STORAGE SECTION.
000410 01 PROGRAM-SWITCHES.
000420     05 WS-DATA-REMAINS-SWITCH    PIC X(3)    VALUE "YES".
000430     05 WS-PREVIOUS-SALESMAN      PIC X(15)   VALUE SPACES.
000440     05 WS-PREVIOUS-LOCATION        PIC X(15)   VALUE SPACES.
000450
000460 01 CONTROL-BREAK-TOTALS.
000470     05 THIS-SALESMAN-TOTAL        PIC 9(6)V99  VALUE ZEROS.
000480     05 THIS-LOCATION-TOTAL          PIC 9(6)V99  VALUE ZEROS.
000490     05 COMPANY-TOTAL              PIC 9(6)V99  VALUE ZEROS.
000500
000510 01 TRANSACTION-WORK-AREA.
000520     05 FILLER                     PIC X(4).
000530     05 TR-SALESMAN-NAME            PIC X(15).
000540     05 TR-ACCOUNT-NUMBER           PIC 9(6).
000550     05 TR-AMOUNT                   PIC 9(4)V99.
000560     05 TR-CODE                     PIC X.
000570     05 TR-RETURNS                   VALUE "R".
000580     05 TR-SALE                     VALUE "S".
000590     05 TR-SALESMAN-LOCATION          PIC X(15).
000600
000610 01 HDG-LINE-ONE.
000620     05 FILLER                     PIC X(15)   VALUE SPACES.
000630     05 FILLER                     PIC X(24)
000640         VALUE "SALES ACTIVITY REPORT - ".
000650     05 HDG-LOCATION                  PIC X(15)   VALUE SPACES.
000660     05 FILLER                     PIC X(78)   VALUE SPACES.
000670
000680 01 HDG-LINE-TWO.
000690     05 FILLER                     PIC X(5)    VALUE SPACES.
000700     05 FILLER                     PIC X(10)   VALUE "SALESMAN: ".
000710     05 HDG-NAME                    PIC X(15).
000720     05 FILLER                     PIC X(25)   VALUE SPACES.
000730     05 FILLER                     PIC X(77)   VALUE SPACES.
000740
000750 01 HDG-LINE-THREE.
000760     05 FILLER                     PIC X(10)   VALUE SPACES.
000770     05 FILLER                     PIC X(11)   VALUE "ACCOUNT #:".
000780     05 FILLER                     PIC X(15)   VALUE SPACES.
000790     05 FILLER                     PIC X(7)    VALUE "RETURNS".
000800     05 FILLER                     PIC X(15)   VALUE SPACES.
000810     05 FILLER                     PIC X(5)    VALUE "SALES".
000820     05 FILLER                     PIC X(69)   VALUE SPACES.

```

FIGURE 8.11 Incorrect two-level control break program

```

000830
000840 01  DETAIL-LINE.
000850      05  FILLER                      PIC X(14)      VALUE SPACES.
000860      05  DET-ACCOUNT-NUMBER          PIC 9(6).
000870      05  FILLER                      PIC X(14)      VALUE SPACES.
000880      05  DET-RETURNS                   PIC $Z,ZZ9.99.
000890      05  FILLER                      PIC X(11)      VALUE SPACES.
000900      05  DET-SALES                     PIC $Z,ZZ9.99.
000910      05  FILLER                      PIC X(69)      VALUE SPACES.
000920
000930 01  SALESMAN-TOTAL-LINE.
000940      05  FILLER                      PIC X(25)      VALUE SPACES.
000950      05  FILLER                      PIC X(21)
000960          VALUE "*** SALESMAN TOTAL = ".
000970      05  PRT-SALESMAN-TOTAL          PIC $Z(3),ZZ9.99CR.
000980      05  FILLER                      PIC X(73)      VALUE SPACES.
000990
001000 01  LOCATION-TOTAL-LINE.
001010      05  FILLER                      PIC X(25)      VALUE SPACES.
001020      05  FILLER                      PIC X(21)
001030          VALUE "*** LOCATION TOTAL = ".
001040      05  PRT-LOCATION-TOTAL            PIC $Z(3),ZZ9.99CR.
001050      05  FILLER                      PIC X(73)      VALUE SPACES.
001060
001070 01  COMPANY-TOTAL-LINE.
001080      05  FILLER                      PIC X(26)      VALUE SPACES.
001090      05  FILLER                      PIC X(20)
001100          VALUE "*** COMPANY TOTAL = ".
001110      05  PRT-COMPANY-TOTAL            PIC $Z(3),ZZ9.99CR.
001120      05  FILLER                      PIC X(73)      VALUE SPACES.
001130
001140  PROCEDURE DIVISION.
001150 010-CALCULATE-CONTROL-BREAKS.
001160      OPEN INPUT SALES-FILE
001170          OUTPUT PRINT-FILE.
001180      READ SALES-FILE INTO TRANSACTION-WORK-AREA
001190          AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
001200      PERFORM 015-PROCESS-ALL-LOCATIONS
001210          UNTIL WS-DATA-REMAINS-SWITCH = "NO".
001220      PERFORM 080-WRITE-COMPANY-TOTAL.
001230      CLOSE SALES-FILE
001240          PRINT-FILE.
001250      STOP RUN.
001260
001270 015-PROCESS-ALL-LOCATIONS.
001280      PERFORM 065-WRITE-LOCATION-HEADING.
001290      MOVE TR-SALESMAN-LOCATION TO WS-PREVIOUS-LOCATION.
001310      PERFORM 020-PROCESS-ALL-SALESMEN
001320          UNTIL TR-SALESMAN-LOCATION NOT EQUAL WS-PREVIOUS-LOCATION
001321              OR WS-DATA-REMAINS-SWITCH = "NO".
001330      PERFORM 075-WRITE-LOCATION-TOTAL.
001340
001350 020-PROCESS-ALL-SALESMEN.
001360      MOVE TR-SALESMAN-NAME TO WS-PREVIOUS-SALESMAN.
001370      MOVE ZEROS TO THIS-SALESMAN-TOTAL.
001380      PERFORM 060-WRITE-SALESMAN-HEADING.
001390      PERFORM 030-PROCESS-ALL-TRANSACTIONS
001400          UNTIL TR-SALESMAN-NAME NOT EQUAL WS-PREVIOUS-SALESMAN
001410              OR WS-DATA-REMAINS-SWITCH = "NO".
001420      PERFORM 070-WRITE-SALESMAN-TOTAL.
001430
001440 030-PROCESS-ALL-TRANSACTIONS.
001450      MOVE SPACES TO DETAIL-LINE.
001460      MOVE TR-ACCOUNT-NUMBER TO DET-ACCOUNT-NUMBER.
001470
001480      IF SALE
001490          MOVE TR-AMOUNT TO DET-SALES
001500          ADD TR-AMOUNT TO THIS-SALESMAN-TOTAL
001510          ADD TR-AMOUNT TO THIS-LOCATION-TOTAL
001520          ADD TR-AMOUNT TO COMPANY-TOTAL
001530      ELSE
001531          IF RETURNS
001540              MOVE TR-AMOUNT TO DET-RETURNS

```

FIGURE 8.11 Continued

```

001550          SUBTRACT TR-AMOUNT FROM THIS-SALESMAN-TOTAL
001560          SUBTRACT TR-AMOUNT FROM THIS-LOCATION-TOTAL
001570          SUBTRACT TR-AMOUNT FROM COMPANY-TOTAL.
001580
001590      WRITE PRINT-LINE FROM DETAIL-LINE
001600          AFTER ADVANCING 1 LINE.
001610      READ SALES-FILE INTO TRANSACTION-WORK-AREA
001620          AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
001630
001640 060-WRITE-SALESMAN-HEADING.
001650      MOVE TR-SALESMAN-NAME TO HDG-NAME.
001660      WRITE PRINT-LINE FROM HDG-LINE-TWO
001670          AFTER ADVANCING 2 LINES.
001680      WRITE PRINT-LINE FROM HDG-LINE-THREE
001690          AFTER ADVANCING 3 LINES.
001700
001710 065-WRITE-LOCATION-HEADING.
001720      MOVE WS-PREVIOUS-LOCATION TO HDG-LOCATION.
001730      WRITE PRINT-LINE FROM HDG-LINE-ONE
001740          AFTER ADVANCING PAGE.
001750
001760 070-WRITE-SALESMAN-TOTAL.
001770      MOVE THIS-SALESMAN-TOTAL TO PRT-SALESMAN-TOTAL.
001780      WRITE PRINT-LINE FROM SALESMAN-TOTAL-LINE
001790          AFTER ADVANCING 2 LINES.
001800
001810 075-WRITE-LOCATION-TOTAL.
001820      MOVE THIS-LOCATION-TOTAL TO PRT-LOCATION-TOTAL.
001830      WRITE PRINT-LINE FROM LOCATION-TOTAL-LINE
001840          AFTER ADVANCING 2 LINES.
001850
001860 080-WRITE-COMPANY-TOTAL.
001870      MOVE COMPANY-TOTAL TO PRT-COMPANY-TOTAL.
001880      WRITE PRINT-LINE FROM COMPANY-TOTAL-LINE
001890          AFTER ADVANCING 5 LINES.

```

FIGURE 8.11 *Continued*

PROJECTS

The personnel office of ABC Electric Company requires summary data on employee salaries. This large corporation has offices in several cities in the nation. Its personnel are also grouped into functional departments, and a given department can appear in more than one city. The employee file has been sorted by location and by department within location. It contains a record for every employee in the company with the following data:

<i>Columns</i>	<i>Field</i>	<i>Picture</i>
1-15	LAST-NAME	X(15)
16-20	SALARY	9(5)
21-23	DEPARTMENT	9(3)
24-35	LOCATION	X(12)

Write a program to compute the total salary for each location, as well as the various department subtotals within a location. Print a detail line showing all information for each employee. Begin the output for each location on a new page. Use the following test data:

```

ADAMS      15000100ATLANTA
BAKER      18000100ATLANTA
CHARLES    17000100ATLANTA
ALLEN      20000200ATLANTA
SMITH      14000200ATLANTA

```

JONES	25000100BOSTON
TYLER	26000250BOSTON
WEBER	18000300BOSTON
WHEELER	14000350BOSTON
GOODMAN	12000100CHICAGO
GORDON	12500100CHICAGO
DAVIS	15000150CHICAGO
ELSWORTH	18000150CHICAGO
HAYWARD	21000150CHICAGO
JACKSON	16000150CHICAGO
BABSON	14000150DETROIT
LEWIS	12000150DETROIT
HAYES	17000300DETROIT
JOHNSON	18000300DETROIT
KELLER	19000300DETROIT

9

SUBPROGRAMS AND THE COPY STATEMENT

OVERVIEW This chapter presents two features to simplify COBOL programming: subprograms and the COPY statement. A *subprogram* is a complete program in its own right, which can be executed from within another COBOL program. Subprograms are used to divide a complex problem into smaller, more manageable units so that several programmers may work on a project simultaneously. They also make it possible for an individual programmer to develop a set of "tools," i.e., programs, which can be reused from project to project.

The COPY statement puts pieces of a program, e.g., record descriptions, into a common library which can be accessed from any COBOL program. It makes previously written code available to any other program without forcing individual programmers to "reinvent the wheel." The COPY statement reduces tedium as well as programmer error.

We begin with a discussion of both features and then develop complete COBOL programs to illustrate the techniques. In so doing, we will review table lookups from Chapter 6 and control breaks from Chapter 8.

COPY STATEMENT Commercial applications are frequently classified into systems, e.g., inventory, accounting, and payroll. Each system in turn consists of several programs, the files of which are interrelated. Indeed, the *same* file is often accessed by several different programs. The COPY clause enables an installation or an individual to build a library of record descriptions and offers the following advantages.

1. Individual programmers need not code the extensive Data Division entries that can make COBOL so tedious. Instead, a programmer codes an appropriate COPY clause. The COBOL compiler then searches a library and brings the proper entries into the COBOL program as though the programmer had written them himself.
2. Changes are made only in one place, i.e., in the library version. Although changes in a file or record description occur infrequently, they do happen. However, only the library version need be altered explicitly, as individual programs will automatically bring in the corrected version during compilation.
3. Programming errors are reduced, and standardization is promoted. Since an individual is coding fewer lines, his program will contain fewer errors. More importantly, all fields are defined correctly. Further, there is no chance of omitting an existing field or erroneously creating a new one. Finally, all programs accessing the same file will use identical record descriptions.

The COPY clause can be used *anywhere* in a COBOL program, *except that the text being copied cannot contain another COPY statement*. The most common use is to bring FDs and/or record descriptions into the Data Division. It is also frequently used in the Procedure Division for entire sections and/or paragraphs. Use of the COPY clause is shown in Figure 9.1.

Several items in Figure 9.1 bear mention. First, consider the COPY statement itself; compiler statement 45 or CEDIT line 000540. (Recall from Chapter 3 *that compiler statement numbers are not the same as CEDIT line numbers*). Compiler statements are numbered consecutively, beginning at one and are always incremented by one. CEDIT line numbers need not be consecutive, and often have increments greater than one.

Compiler statement numbers (consecutive)

CEDIT line numbers

39	000480				
40	000490	01	CONTROL -BREAK-TOTALS.		
41	000500	05	THIS-SALESMAN-TOTAL	PIC S9(6)V99	VALUE ZEROS.
42	000510	05	THIS-LOCATION-TOTAL	PIC S9(6)V99	VALUE ZEROS.
43	000520	05	COMPANY-TOTAL	PIC S9(6)V99	VALUE ZEROS.
44	000530				
45	000540		COPY "SALESMAN/CBL".		COPY statement
46	000001	01	TRANSACTION-WORK-AREA.		
47	000002	05	FILLER	PIC X(4).	
48	000003	05	TR-SALESMAN-NAME	PIC X(15).	
49	000004	05	TR-ACCOUNT-NUMBER	PIC 9(6).	
50	000005	05	TR-AMOUNT	PIC 9(4)V99.	
51	000006	05	TR-CODE	PIC X.	
52	000007		88 RETURNS	VALUE "R".	
53	000008		88 SALE	VALUE "S".	
54	000009	05	TR-LOCATION-CODE	PIC X(3).	
55	000010	05	FILLER	PIC X(12).	
56	000550				
57	000560	01	HDG-LINE-ONE.		
58	000570	05	FILLER	PIC X(15)	VALUE SPACES.
59	000580	05	FILLER	PIC X(24)	
60	000590		VALUE "SALES ACTIVITY REPORT - "		
61	000600	05	HDG-LOCATION	PIC X(15)	VALUE SPACES.

These entries are brought into the program by the COPY statement

FIGURE 9.1 The COPY statement

Figure 9.1 combines statements from *two* CEDIT files into a single program. The first file contains the COPY statement of compiler line 45 (CEDIT line number 540). The COPY statement brings in 10 lines from a second file, SALESMAN/CBL as though those statements appeared in the original program. *Note that the compiler statement numbers are consecutive, whereas the CEDIT line numbers are not.* The CEDIT line numbers run from 480 to 540, 1 to 10, and 550 to 600. The compiler statement numbers go from 39 to 61. In other words, one file contains CEDIT line numbers 480–540 and 550–600. The *copied* entry contains CEDIT line numbers 1–10. When the COBOL compiler encounters the COPY statement at line 540, it brings in CEDIT line numbers 1–10 from the second file, SALESMAN/CBL. (The code in Figure 9.1 has been extracted from a complete program which appears later in the chapter.)

SUBPROGRAMS

A *subprogram* is a *complete* COBOL program which receives control at *execution* time. Subprograms are independent of any other program, and contain the four divisions of a regular program. In addition a subprogram contains a LINKAGE SECTION in its Data Division, that passes information to and from the main program. The same program may call several subprograms, and a subprogram may in turn call another subprogram.

Consider Figure 9.2, which depicts skeletal code illustrating the linkage between a main (or calling) and sub (or called) program. (Figures 9.4 and 9.5 at the chapter's end, contain complete COBOL programs to further illustrate these points.) The main program contains a CALL statement somewhere in its Procedure Division. When the CALL is executed, control is transferred to the first statement in the Procedure Division of the subprogram. The latter continues executing until it encounters an EXIT PROGRAM statement, at which point control returns to the main program to the

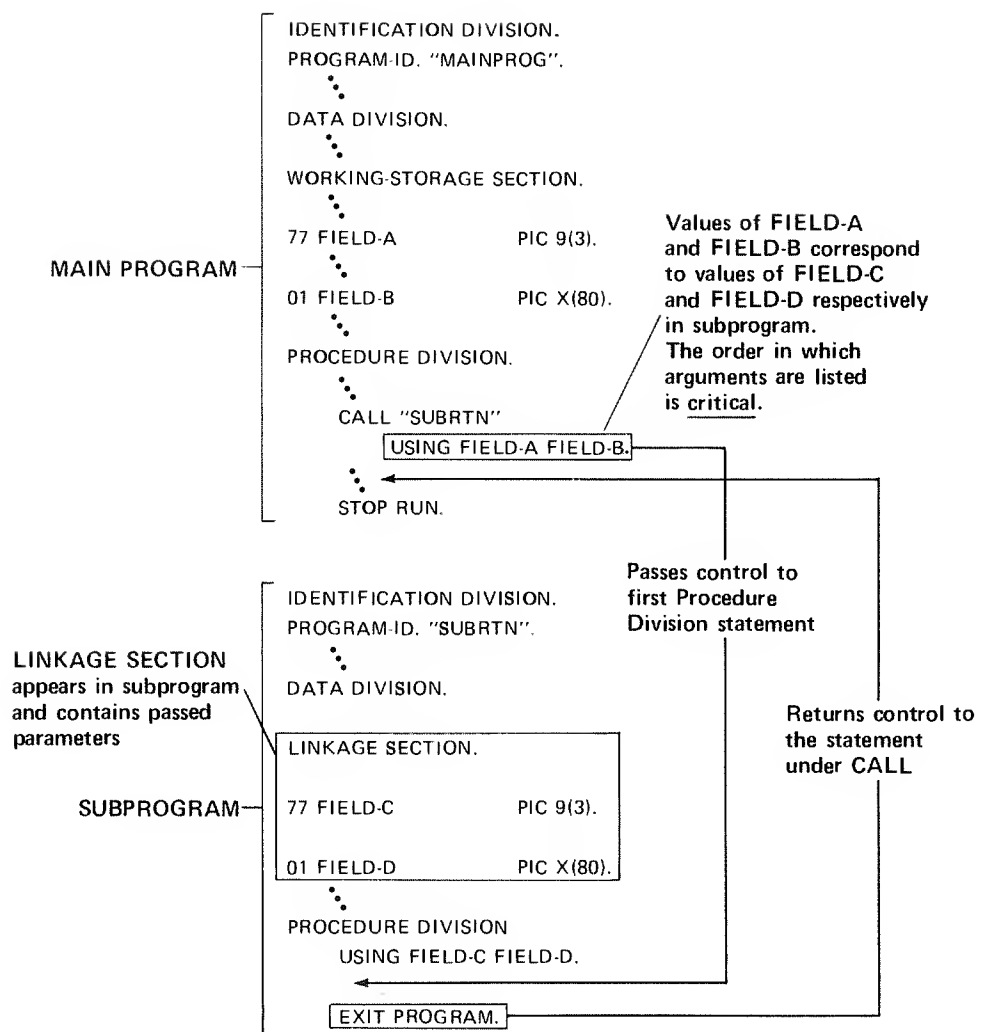


FIGURE 9.2 Skeletal code for a subprogram

statement immediately following the initial CALL. The EXIT PROGRAM statement must be in a paragraph by itself.

Data are passed between the main and subprogram via USING clauses, which appear in the CALL statement of the main program and in the Procedure Division header of the subprogram. Data names that are passed as arguments must be defined in the Linkage Section of the subprogram. Any CALL statement in the main program contains a USING clause that specifies the data on which the subprogram is to operate; for example, CALL "SUBRTN" USING FIELD-A FIELD-B. The subprogram in turn contains a USING clause in its Procedure Division header, for example, PROCEDURE DIVISION USING FIELD-C FIELD-D.

The data names in the main and subprogram USING clauses are different, but the *order* of data names within these clauses is critical. The first item in the USING clause of the main program, FIELD-A, corresponds to the first item in the USING clause of the subprogram, FIELD-C. Both are defined as three-position numeric fields. In similar fashion, FIELD-B of the main program corresponds to FIELD-D of the subprogram. Note well that as either program changes the value of a passed parameter, that value changes simultaneously in both programs. This is because only a single

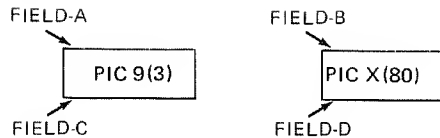


FIGURE 9.3 Storage allocation for passed parameters

storage location is assigned to both data names; for example, the same location is assigned for FIELD-A from the main program and FIELD-C from the subprogram. This situation is illustrated in Figure 9.3.

As can be seen from Figure 9.3, FIELD-A and FIELD-C reference the *same* locations in memory. In similar fashion FIELD-B and FIELD-D also reference a common area. The COBOL compiler allocates the *same* space for FIELD-A and FIELD-C, or FIELD-B and FIELD-D.

The parameters of the subprogram, FIELD-C and FIELD-D, must be defined in the Linkage Section of the subprogram. In other words, any data name appearing in a Linkage Section already has had space allocated, i.e., by the program which called it. Hence, the space for FIELD-C and FIELD-D was previously assigned in the main program to FIELD-A and FIELD-B, respectively. The USING clauses in both programs establish the correspondence between the data names.

Finally, all passed parameters must be defined as either 01 or 77-level entries. *The order in which arguments are listed is critical.* Passing parameters in the wrong sequence is one of the most frequent problems in modular systems consisting of several programs.

A COMPLETE EXAMPLE

We now develop two programs to further illustrate the COPY statement and subprograms. We will expand the double control break problem of Chapter 8, and review the table lookup procedure of Chapter 6. Specifications are as follows:

Process the salesfile of Chapter 8 (Figure 8.6a) to produce two sets of totals; on location and salesman within location. This time, however, the incoming record contains a three-position location code which is to be expanded in the subprogram. In addition, the description of the incoming record is to be established via a COPY clause. The intended output is identical to that of Figure 8.6b.

Figure 9.4 contains the main (calling) program and Figure 9.5 the sub (called) program. The main program is very similar to the two-level program of Chapter 8 (Figure 8.9) with two exceptions. First, the incoming record is *copied* into the program (line 540) and contains a location code rather than an expanded location name. Second, the location code is expanded in a subprogram.

Note well the COPY statement of line 540. It references the file SALESMAN/CBL (which must exist on a diskette), and which contains the 10 lines of TRANSACTION-WORK-AREA. The latter is copied into the main program, after which the CREDIT line numbers of the main program resume at 550.

Lines 1720-1740 of the main program contain a CALL statement which transfers control to the first executable statement of the subprogram. (The *object* program DECODER/COB must also exist on a diskette.) The USING clause in the main program contains two parameters, PASSED-LOCATION-CODE and PASSED-EXPANDED-LOCATION which correspond to the two parameters in the USING clause of the subprogram (lines

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID. MAINPROG.
000120 AUTHOR. R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER. TRS-80.
000170 OBJECT-COMPUTER. TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210     SELECT SALES-FILE
000220     ASSIGN TO INPUT "ONELEVEL/DAT".
000230     SELECT PRINT-FILE
000240     ASSIGN TO PRINT "MAINPROG/TXT".
000250
000260 DATA DIVISION.
000270 FILE SECTION.
000280 FD SALES-FILE
000290     LABEL RECORDS ARE STANDARD
000300     RECORD CONTAINS 47 CHARACTERS
000310     DATA RECORD IS SALES-RECORD.
000320 01 SALES-RECORD PIC X(47).
000330
000340 FD PRINT-FILE
000350     LABEL RECORDS ARE STANDARD
000360     RECORD CONTAINS 132 CHARACTERS
000370     DATA RECORD IS PRINT-LINE.
000380 01 PRINT-LINE PIC X(132).
000390
000400 WORKING-STORAGE SECTION.
000410 77 PASSED-LOCATION-CODE PIC X(3).
000420 77 PASSED-EXPANDED-LOCATION PIC X(15).
000430
000440 01 PROGRAM-SWITCHES.
000450 05 WS-DATA-REMAINS-SWITCH PIC X(3) VALUE "YES".
000460 05 WS-PREVIOUS-SALESMAN PIC X(15) VALUE SPACES.
000470 05 WS-PREVIOUS-LOCATION-CODE PIC X(3) VALUE SPACES.
000480
000490 01 CONTROL-BREAK-TOTALS.
000500 05 THIS-SALESMAN-TOTAL PIC S9(6)V99 VALUE ZEROS.
000510 05 THIS-LOCATION-TOTAL PIC S9(6)V99 VALUE ZEROS.
000520 05 COMPANY-TOTAL PIC S9(6)V99 VALUE ZEROS.
000530
000540 COPY "SALESMAN/CBL". COPY clause
000001 01 TRANSACTION-WORK-AREA.
000002 05 FILLER PIC X(4).
000003 05 TR-SALESMAN-NAME PIC X(15).
000004 05 TR-ACCOUNT-NUMBER PIC 9(6).
000005 05 TR-AMOUNT PIC 9(4)V99.
000006 05 TR-CODE PIC X.
000007 88 RETURNS VALUE "R".
000008 88 SALE VALUE "S".
000009 05 TR-LOCATION-CODE PIC X(3).
000010 05 FILLER PIC X(12).
000550
000560 01 HDG-LINE-ONE.
000570 05 FILLER PIC X(15) VALUE SPACES.
000580 05 FILLER PIC X(24)
000590     VALUE "SALES ACTIVITY REPORT - ".
000600 05 HDG-LOCATION PIC X(15) VALUE SPACES.
000610 05 FILLER PIC X(78) VALUE SPACES.
000620
000630 01 HDG-LINE-TWO.
000640 05 FILLER PIC X(5) VALUE SPACES.
000650 05 FILLER PIC X(10) VALUE "SALESMAN: ".
000660 05 HDG-NAME PIC X(15).
000670 05 FILLER PIC X(25) VALUE SPACES.
000680 05 FILLER PIC X(77) VALUE SPACES.
000690
000700 01 HDG-LINE-THREE.
000710 05 FILLER PIC X(10) VALUE SPACES.
000720 05 FILLER PIC X(11) VALUE "ACCOUNT #: ".

```

Copied line numbers are
out of sequence

FIGURE 9.4 The calling (main) program

```

000730      05  FILLER                                PIC X(15)      VALUE SPACES.
000740      05  FILLER                                PIC X(7)       VALUE "RETURNS".
000750      05  FILLER                                PIC X(15)      VALUE SPACES.
000760      05  FILLER                                PIC X(5)       VALUE "SALES".
000770      05  FILLER                                PIC X(69)      VALUE SPACES.
000780
000790 01  DETAIL-LINE.
000800      05  FILLER                                PIC X(14)      VALUE SPACES.
000810      05  DET-ACCOUNT-NUMBER                      PIC 9(6).
000820      05  FILLER                                PIC X(14)      VALUE SPACES.
000830      05  DET-RETURNS                            PIC $Z,ZZ9.99.
000840      05  FILLER                                PIC X(11)      VALUE SPACES.
000850      05  DET-SALES                              PIC $Z,ZZ9.99.
000860      05  FILLER                                PIC X(69)      VALUE SPACES.
000870
000880 01  SALESMAN-TOTAL-LINE.
000890      05  FILLER                                PIC X(25)      VALUE SPACES.
000900      05  FILLER                                PIC X(21)
000910          VALUE "*** SALESMAN TOTAL = ".
000920      05  PRT-SALESMAN-TOTAL                    PIC $Z(3),ZZ9.99CR.
000930      05  FILLER                                PIC X(73)      VALUE SPACES.
000940
000950 01  LOCATION-TOTAL-LINE.
000960      05  FILLER                                PIC X(25)      VALUE SPACES.
000970      05  FILLER                                PIC X(21)
000980          VALUE "*** LOCATION TOTAL = ".
000990      05  PRT-LOCATION-TOTAL                      PIC $Z(3),ZZ9.99CR.
001000      05  FILLER                                PIC X(73)      VALUE SPACES.
001010
001020 01  COMPANY-TOTAL-LINE.
001030      05  FILLER                                PIC X(26)      VALUE SPACES.
001040      05  FILLER                                PIC X(20)
001050          VALUE "*** COMPANY TOTAL = ".
001060      05  PRT-COMPANY-TOTAL                      PIC $Z(3),ZZ9.99CR.
001070      05  FILLER                                PIC X(73)      VALUE SPACES.
001080
001090 PROCEDURE DIVISION.
001100 010-CALCULATE-CONTROL-BREAKS.
001110     OPEN INPUT SALES-FILE
001120     OUTPUT PRINT-FILE.
001130     READ SALES-FILE INTO TRANSACTION-WORK-AREA
001140     AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
001150     PERFORM 015-PROCESS-ALL-LOCATIONS
001160     UNTIL WS-DATA-REMAINS-SWITCH = "NO".
001170     PERFORM 080-WRITE-COMPANY-TOTAL.
001180     CLOSE SALES-FILE
001190     PRINT-FILE.
001200     STOP RUN.
001210
001220 015-PROCESS-ALL-LOCATIONS.
001230     PERFORM 065-WRITE-LOCATION-HEADING.
001240     MOVE TR-LOCATION-CODE TO WS-PREVIOUS-LOCATION-CODE.
001250     MOVE ZEROS TO THIS-LOCATION-TOTAL.
001260     PERFORM 020-PROCESS-ALL-SALESMEN
001270     UNTIL TR-LOCATION-CODE NOT EQUAL WS-PREVIOUS-LOCATION-CODE
001280     OR WS-DATA-REMAINS-SWITCH = "NO".
001290     PERFORM 075-WRITE-LOCATION-TOTAL.
001300
001310 020-PROCESS-ALL-SALESMEN.
001320     MOVE TR-SALESMAN-NAME TO WS-PREVIOUS-SALESMAN.
001330     MOVE ZEROS TO THIS-SALESMAN-TOTAL.
001340     PERFORM 060-WRITE-SALESMAN-HEADING.
001350     PERFORM 030-PROCESS-ALL-TRANSACTIONS
001360     UNTIL TR-SALESMAN-NAME NOT EQUAL WS-PREVIOUS-SALESMAN
001370     OR WS-DATA-REMAINS-SWITCH = "NO".
001380     PERFORM 070-WRITE-SALESMAN-TOTAL.
001390
001400 030-PROCESS-ALL-TRANSACTIONS.
001410     MOVE SPACES TO DETAIL-LINE.
001420     MOVE TR-ACCOUNT-NUMBER TO DET-ACCOUNT-NUMBER.
001430
001440     IF SALE
001450         MOVE TR-AMOUNT TO DET-SALES

```

FIGURE 9.4 *Continued*

```

001460      ADD TR-AMOUNT TO THIS-SALESMAN-TOTAL
001470      ADD TR-AMOUNT TO THIS-LOCATION-TOTAL
001480      ADD TR-AMOUNT TO COMPANY-TOTAL
001490      ELSE
001500          IF RETURNS
001510              MOVE TR-AMOUNT TO DET-RETURNS
001520              SUBTRACT TR-AMOUNT FROM THIS-SALESMAN-TOTAL
001530              SUBTRACT TR-AMOUNT FROM THIS-LOCATION-TOTAL
001540              SUBTRACT TR-AMOUNT FROM COMPANY-TOTAL.
001550
001560      WRITE PRINT-LINE FROM DETAIL-LINE
001570          AFTER ADVANCING 1 LINE.
001580      READ SALES-FILE INTO TRANSACTION-WORK-AREA
001590          AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
001600
001610 060-WRITE-SALESMAN-HEADING.
001620      MOVE TR-SALESMAN-NAME TO HDG-NAME.
001630      WRITE PRINT-LINE FROM HDG-LINE-TWO
001640          AFTER ADVANCING 2 LINES.
001650      WRITE PRINT-LINE FROM HDG-LINE-THREE
001660          AFTER ADVANCING 3 LINES.
001670
001680 065-WRITE-LOCATION-HEADING.
001690      MOVE TR-LOCATION-CODE TO PASSED-LOCATION-CODE.
001700      MOVE SPACES TO PASSED-EXPANDED-LOCATION.
001710
001720      CALL "DECODER/COB" USING
001730          PASSED-LOCATION-CODE
001740          PASSED-EXPANDED-LOCATION.
001750
001760      MOVE PASSED-EXPANDED-LOCATION TO HDG-LOCATION.
001770      WRITE PRINT-LINE FROM HDG-LINE-ONE
001780          AFTER ADVANCING PAGE.
001790
001800 070-WRITE-SALESMAN-TOTAL.
001810      MOVE THIS-SALESMAN-TOTAL TO PRT-SALESMAN-TOTAL.
001820      WRITE PRINT-LINE FROM SALESMAN-TOTAL-LINE
001830          AFTER ADVANCING 2 LINES.
001840
001850 075-WRITE-LOCATION-TOTAL.
001860      MOVE THIS-LOCATION-TOTAL TO PRT-LOCATION-TOTAL.
001870      WRITE PRINT-LINE FROM LOCATION-TOTAL-LINE
001880          AFTER ADVANCING 2 LINES.
001890
001900 080-WRITE-COMPANY-TOTAL.
001910      MOVE COMPANY-TOTAL TO PRT-COMPANY-TOTAL.
001920      WRITE PRINT-LINE FROM COMPANY-TOTAL-LINE
001930          AFTER ADVANCING 5 LINES.

```

Call to subprogram

FIGURE 9.4 *Continued*

490 and 500 of Figure 9.5). The parameters of the subprogram INCOMING-LOCATION-CODE and EXPANDED-PRINT-LOCATION, are defined in the Linkage Section indicating that space for these data names has already been allocated.

The subprogram begins execution on line 520. It executes statements in much the same way as any other COBOL program. However, when the subprogram is finished, it encounters an EXIT PROGRAM statement (line 590) which returns control to the main program to the line under the original CALL.

A word or two is in order regarding the nature of the subprogram itself. Its job is to expand a three-position location code, INCOMING-LOCATION-CODE, to a 15-position expanded value, EXPANDED-PRINT-LOCATION. The table of codes and corresponding expanded values is defined in lines 270-420 of the subprogram. Observe the relationship between the VALUE, OCCURS, and REDEFINES clauses as originally explained in Chapter 6.

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID. DECODER.
000120 AUTHOR. R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER. TRS-80.
000170 OBJECT-COMPUTER. TRS-80.
000180
000190 DATA DIVISION.
000200 WORKING-STORAGE SECTION.
000210 01 PROGRAM-SWITCHES.
000220 05 WS-LOCATION-SWITCH PIC X(3) VALUE SPACES.
000230
000240 01 PROGRAM-SUBSCRIPTS.
000250 05 LOCATION-SUB PIC 99.
000260
000270 01 LOCATION-VALUES.
000280 05 FILLER PIC X(18) VALUE "ATLANTA" ".
000290 05 FILLER PIC X(18) VALUE "BOSTON" ".
000300 05 FILLER PIC X(18) VALUE "CHICAGO" ".
000310 05 FILLER PIC X(18) VALUE "DETROIT" ".
000320 05 FILLER PIC X(18) VALUE "LOUISVILLE" ".
000330 05 FILLER PIC X(18) VALUE "MINNEAPOLIS" ".
000340 05 FILLER PIC X(18) VALUE "NEWARK" ".
000350 05 FILLER PIC X(18) VALUE "NY NEW YORK" ".
000360 05 FILLER PIC X(18) VALUE "SA SAN ANTONIO" ".
000370 05 FILLER PIC X(18) VALUE "SF SAN FRANCISCO".
000380
000390 01 LOCATION-TABLE REDEFINES LOCATION-VALUES.
000400 05 LOCATION-CODE-AND-VALUE OCCURS 10 TIMES.
000410 10 LOCATION-CODE PIC X(3).
000420 10 EXPANDED-LOCATION PIC X(15).
000430
000440 LINKAGE SECTION.
000450 01 INCOMING-LOCATION-CODE PIC X(3).
000460 01 EXPANDED-PRINT-LOCATION PIC X(15).
000470
000480 PROCEDURE DIVISION
000490 USING INCOMING-LOCATION-CODE
000500 EXPANDED-PRINT-LOCATION.
000510
000520 400-MAINLINE.
000530 MOVE "NO" TO WS-LOCATION-SWITCH.
000540 PERFORM 500-EXPAND-LOCATION-CODE
000550 VARYING LOCATION-SUB FROM 1 BY 1
000560 UNTIL WS-LOCATION-SWITCH = "YES".
000570
000580 450-RETURN-TO-MAIN.
000590 EXIT PROGRAM.
000600
000610 500-EXPAND-LOCATION-CODE.
000620 IF LOCATION-SUB > 10
000630 MOVE "UNKNOWN" TO EXPANDED-PRINT-LOCATION
000640 MOVE "YES" TO WS-LOCATION-SWITCH
000650 ELSE
000660 IF INCOMING-LOCATION-CODE = LOCATION-CODE (LOCATION-SUB)
000670 MOVE EXPANDED-LOCATION (LOCATION-SUB)
000680 TO EXPANDED-PRINT-LOCATION
000690 MOVE "YES" TO WS-LOCATION-SWITCH.
000700

```

Linkage Section is in the called program

USING clause is part of the
Procedure Division header

One statement paragraph returns control to main program

FIGURE 9.5 The sub (called) program

The table lookup procedure itself moves "NO" to a switch (line 530) and then sequentially steps through the table of codes until a match is found. (See the PERFORM/VARYING of lines 540-560.) Note well the check in line 620 for an invalid code. Observe also that "YES" is moved to the previously referenced switch to terminate the search for either an invalid code (line 640) or a match (line 690).

SUMMARY This chapter dealt with subprograms and the COPY statement. Both techniques were discussed and illustrated in complete programs. In addition, the table lookup procedure of Chapter 6 was reviewed.

Before leaving the subject of subprograms, we offer one programming tip regarding the *order* of arguments in the USING clauses. Recall that *the order in which parameters are listed is critical*, and further that the picture clauses must be identical. One can guarantee that these conditions are met by passing a *single* 01 record that is *copied* into both programs. Consider Figure 9.6, which illustrates both "poor" and "improved" code for passing parameters.

Use of the same COPY clause in both programs eliminates any problem with listing arguments in the wrong order and/or inconsistent definition through different picture clauses. In addition, passing only a *single* 01 parameter facilitates coding in the USING clauses and makes them immune to change.

Poor Code:

```
CALL "DECODER/COB"
  USING TITLE-CODE, EXPANDED-TITLE,
        LOCATION-CODE, EXPANDED-LOCATION.
.
.
PROCEDURE DIVISION
  USING LS-TITLE-CODE, LS-EXPANDED-TITLE,
        LS-LOCATION-CODE, LS-EXPANDED-LOCATION.
```

Improved Code:

```
COPY "ARGUMENTS/CBL".

01 PARAMETER-LIST.
05 TITLE-CODE          PIC 9(3).
05 EXPANDED-TITLE      PIC X(15).
05 LOCATION-CODE       PIC XX.
05 EXPANDED-LOCATION     PIC X(12).
.
.
CALL "DECODER"
  USING PARAMETER-LIST.

LINKAGE SECTION.
COPY "ARGUMENTS/CBL".

01 PARAMETER-LIST.
05 TITLE-CODE          PIC 9(3).
05 EXPANDED-TITLE      PIC X(15).
05 LOCATION-CODE       PIC XX.
05 EXPANDED-LOCATION     PIC X(12).

PROCEDURE DIVISION
  USING PARAMETER-LIST.
```

These entries will be copied from a common file

FIGURE 9.6 Passing parameters to a subprogram

TRUE/FALSE

1. The Linkage Section appears in the subprogram.
2. Data names in 'CALL...USING' and 'PROCEDURE DIVISION...USING' must be the same.
3. The Linkage Section appears in the calling program.
4. A subprogram contains only the Data and Procedure Divisions.
5. A called program may call another program.
6. The COPY clause can be used in the Procedure Division.
7. The COPY statement cannot be used to initialize a table.
8. Data names passed to a subprogram are limited to 01 or 77-level entries.
9. The order of arguments in a USING clause is unimportant.
10. A single program may contain multiple COPY statements.

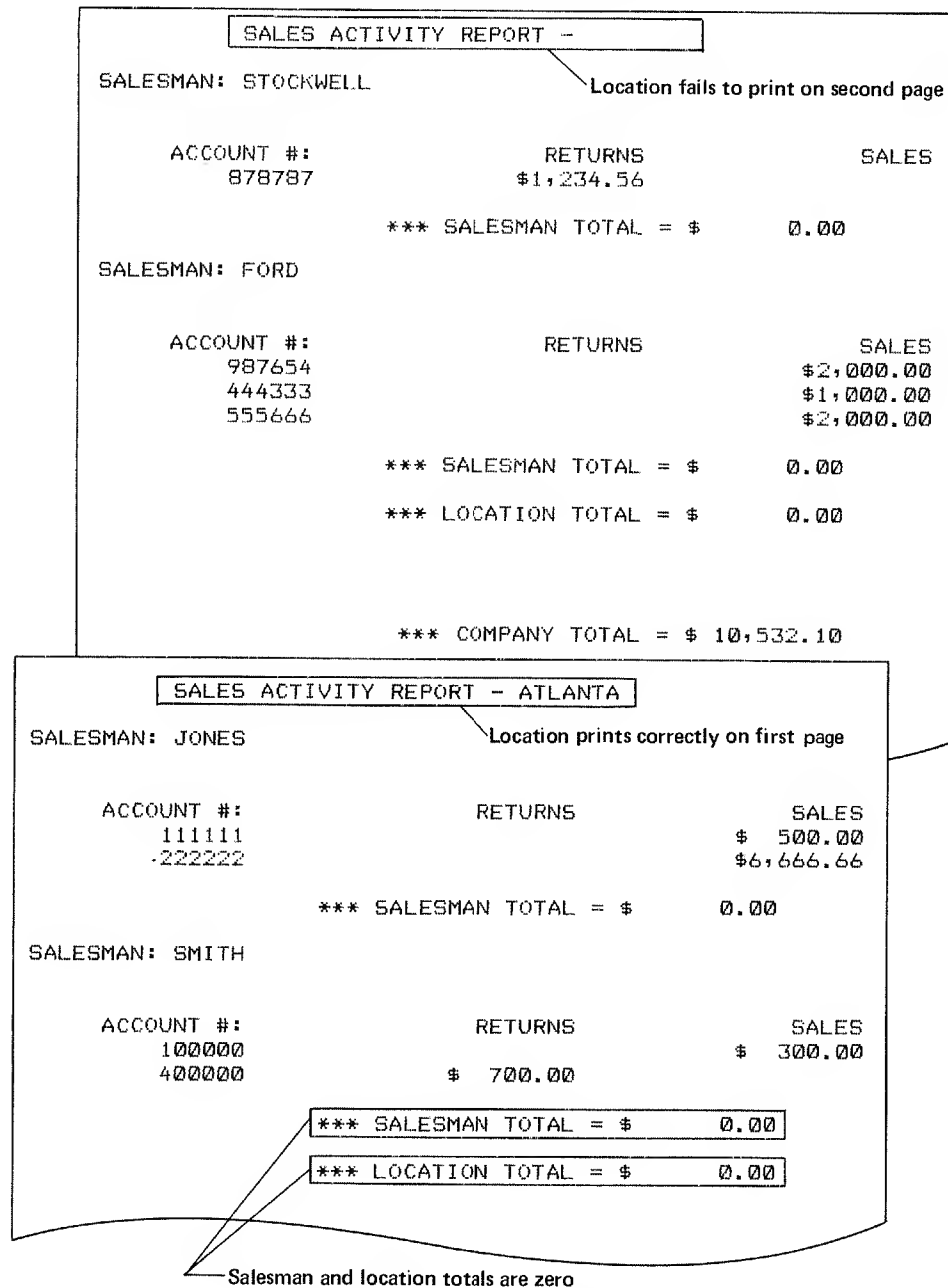


FIGURE 9.7 Invalid output of main and subprogram
(see Exercise 2 on page 178)

EXERCISES

1. State the advantages, if any, of using a COPY statement rather than "hard coding" a table in the program. Describe the necessary steps to modify a table that has been defined in a COPY statement *after* the initial program has been put into production.
2. *Debugging*: Figure 9.7 contains invalid output produced by the main and subprograms of Figures 9.8 and 9.9, respectively. The input and intended output correspond to Figures 8.6a and b. (Figures 9.8 and 9.9 are modified versions of the programs in Figures 9.4 and 9.5, which appeared in the chapter.)
Find and correct all errors in both the main and subprograms.

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID.    EMAINPRO.
000120 AUTHOR.        R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER.    TRS-80.
000170 OBJECT-COMPUTER.    TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210     SELECT SALES-FILE
000220         ASSIGN TO INPUT "ONELEVEL/DAT".
000230     SELECT PRINT-FILE
000240         ASSIGN TO PRINT "EMAINPRO/TXT".
000250
000260 DATA DIVISION.
000270 FILE SECTION.
000280 FD  SALES-FILE
000290     LABEL RECORDS ARE STANDARD
000300     RECORD CONTAINS 47 CHARACTERS
000310     DATA RECORD IS SALES-RECORD.
000320 01  SALES-RECORD                PIC X(47).
000330
000340 FD  PRINT-FILE
000350     LABEL RECORDS ARE STANDARD
000360     RECORD CONTAINS 132 CHARACTERS
000370     DATA RECORD IS PRINT-LINE.
000380 01  PRINT-LINE                  PIC X(132).
000390
000400 WORKING-STORAGE SECTION.
000410 77  PASSED-LOCATION-CODE          PIC X(3).
000420 77  PASSED-EXPANDED-LOCATION     PIC X(15).
000430
000440 01  PROGRAM-SWITCHES.
000450     05  WS-DATA-REMAINS-SWITCH   PIC X(3)   VALUE "YES".
000460     05  WS-PREVIOUS-SALESMAN     PIC X(15)  VALUE SPACES.
000470     05  WS-PREVIOUS-LOCATION-CODE PIC X(3)   VALUE SPACES.
000480
000490 01  CONTROL-BREAK-TOTALS.
000500     05  THIS-SALESMAN-TOTAL      PIC S9(6)V99 VALUE ZEROS.
000510     05  THIS-LOCATION-TOTAL       PIC S9(6)V99 VALUE ZEROS.
000520     05  COMPANY-TOTAL           PIC S9(6)V99 VALUE ZEROS.
000530
000540     COPY "SALESMAN/CBL".
000001 01  TRANSACTION-WORK-AREA.
000002     05  FILLER                PIC X(4).
000003     05  TR-SALESMAN-NAME        PIC X(15).
000004     05  TR-ACCOUNT-NUMBER       PIC 9(6).
000005     05  TR-AMOUNT              PIC 9(4)V99.
000006     05  TR-CODE                PIC X.
000007     88  RETURNS                 VALUE "R".
000008     88  SALE                     VALUE "S".

```

FIGURE 9.8 Invalid main program

```

000009      05  TR-LOCATION-CODE      PIC X(3).
000010      05  FILLER                PIC X(12).
000550
000560 01  HDG-LINE-ONE.
000570      05  FILLER                PIC X(15)      VALUE SPACES.
000580      05  FILLER                PIC X(24)
000590          VALUE "SALES ACTIVITY REPORT - ".
000600      05  HDG-LOCATION            PIC X(15)      VALUE SPACES.
000610      05  FILLER                PIC X(78)      VALUE SPACES.
000620
000630 01  HDG-LINE-TWO.
000640      05  FILLER                PIC X(5)       VALUE SPACES.
000650      05  FILLER                PIC X(10)      VALUE "SALESMAN: ".
000660      05  HDG-NAME              PIC X(15).
000670      05  FILLER                PIC X(25)      VALUE SPACES.
000680      05  FILLER                PIC X(77)      VALUE SPACES.
000690
000700 01  HDG-LINE-THREE.
000710      05  FILLER                PIC X(10)      VALUE SPACES.
000720      05  FILLER                PIC X(11)      VALUE "ACCOUNT #: ".
000730      05  FILLER                PIC X(15)      VALUE SPACES.
000740      05  FILLER                PIC X(7)       VALUE "RETURNS".
000750      05  FILLER                PIC X(15)      VALUE SPACES.
000760      05  FILLER                PIC X(5)       VALUE "SALES".
000770      05  FILLER                PIC X(69)      VALUE SPACES.
000780
000790 01  DETAIL-LINE.
000800      05  FILLER                PIC X(14)      VALUE SPACES.
000810      05  DET-ACCOUNT-NUMBER    PIC 9(6).
000820      05  FILLER                PIC X(14)      VALUE SPACES.
000830      05  DET-RETURNS          PIC $Z,ZZ9.99.
000840      05  FILLER                PIC X(11)      VALUE SPACES.
000850      05  DET-SALES             PIC $Z,ZZ9.99.
000860      05  FILLER                PIC X(69)      VALUE SPACES.
000870
000880 01  SALESMAN-TOTAL-LINE.
000890      05  FILLER                PIC X(25)      VALUE SPACES.
000900      05  FILLER                PIC X(21)
000910          VALUE "*** SALESMAN TOTAL = ".
000920      05  PRT-SALESMAN-TOTAL   PIC $Z(3),ZZ9.99CR.
000930      05  FILLER                PIC X(73)      VALUE SPACES.
000940
000950 01  LOCATION-TOTAL-LINE.
000960      05  FILLER                PIC X(25)      VALUE SPACES.
000970      05  FILLER                PIC X(21)
000980          VALUE "*** LOCATION TOTAL = ".
000990      05  PRT-LOCATION-TOTAL      PIC $Z(3),ZZ9.99CR.
001000      05  FILLER                PIC X(73)      VALUE SPACES.
001010
001020 01  COMPANY-TOTAL-LINE.
001030      05  FILLER                PIC X(26)      VALUE SPACES.
001040      05  FILLER                PIC X(20)
001050          VALUE "*** COMPANY TOTAL = ".
001060      05  PRT-COMPANY-TOTAL      PIC $Z(3),ZZ9.99CR.
001070      05  FILLER                PIC X(73)      VALUE SPACES.
001080
001090 010-PROCEDURE DIVISION.
001100 010-CALCULATE-CONTROL-BREAKS.
001110     OPEN INPUT SALES-FILE
001120     OUTPUT PRINT-FILE.
001130     READ SALES-FILE INTO TRANSACTION-WORK-AREA
001140     AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
001150     PERFORM 015-PROCESS-ALL-LOCATIONS
001160     UNTIL WS-DATA-REMAINS-SWITCH = "NO".
001170     PERFORM 080-WRITE-COMPANY-TOTAL.
001180     CLOSE SALES-FILE
001190     PRINT-FILE.
001200     STOP RUN.
001210
001220 015-PROCESS-ALL-LOCATIONS.

```

FIGURE 9.8 *Continued*

```

001230    PERFORM 065-WRITE-LOCATION-HEADING.
001240    MOVE TR-LOCATION-CODE TO WS-PREVIOUS-LOCATION-CODE.
001250    MOVE ZEROS TO THIS-LOCATION-TOTAL.
001260    PERFORM 020-PROCESS-ALL-SALESMEN
001270        UNTIL TR-LOCATION-CODE NOT EQUAL WS-PREVIOUS-LOCATION-CODE
001280            OR WS-DATA-REMAINS-SWITCH = "NO".
001290    PERFORM 075-WRITE-LOCATION-TOTAL.
001300
001310 020-PROCESS-ALL-SALESMEN.
001320    MOVE TR-SALESMAN-NAME TO WS-PREVIOUS-SALESMAN.
001330    MOVE ZEROS TO THIS-SALESMAN-TOTAL.
001340    PERFORM 060-WRITE-SALESMAN-HEADING.
001350    PERFORM 030-PROCESS-ALL-TRANSACTIONS
001360        UNTIL TR-SALESMAN-NAME NOT EQUAL WS-PREVIOUS-SALESMAN
001370            OR WS-DATA-REMAINS-SWITCH = "NO".
001380    PERFORM 070-WRITE-SALESMAN-TOTAL.
001390
001400 030-PROCESS-ALL-TRANSACTIONS.
001410    MOVE SPACES TO DETAIL-LINE.
001420    MOVE TR-ACCOUNT-NUMBER TO DET-ACCOUNT-NUMBER.
001430
001440    IF SALE
001450        MOVE TR-AMOUNT TO DET-SALES
001460*        ADD TR-AMOUNT TO THIS-SALESMAN-TOTAL
001470*        ADD TR-AMOUNT TO THIS-LOCATION-TOTAL
001480        ADD TR-AMOUNT TO COMPANY-TOTAL
001490    ELSE
001500        IF RETURNS
001510            MOVE TR-AMOUNT TO DET-RETURNS
001520*            SUBTRACT TR-AMOUNT FROM THIS-SALESMAN-TOTAL
001530*            SUBTRACT TR-AMOUNT FROM THIS-LOCATION-TOTAL
001540            SUBTRACT TR-AMOUNT FROM COMPANY-TOTAL.
001550
001560    WRITE PRINT-LINE FROM DETAIL-LINE
001570        AFTER ADVANCING 1 LINE.
001580    READ SALES-FILE INTO TRANSACTION-WORK-AREA
001590        AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
001600
001610 060-WRITE-SALESMAN-HEADING.
001620    MOVE TR-SALESMAN-NAME TO HDG-NAME.
001630    WRITE PRINT-LINE FROM HDG-LINE-TWO
001640        AFTER ADVANCING 2 LINES.
001650    WRITE PRINT-LINE FROM HDG-LINE-THREE
001660        AFTER ADVANCING 3 LINES.
001670
001680 065-WRITE-LOCATION-HEADING.
001690    MOVE TR-LOCATION-CODE TO PASSED-LOCATION-CODE.
001700    MOVE SPACES TO PASSED-EXPANDED-LOCATION.
001710
001720    CALL "EDECODER/COB" USING
001730        PASSED-LOCATION-CODE
001740        PASSED-EXPANDED-LOCATION.
001750
001760    MOVE PASSED-EXPANDED-LOCATION TO HDG-LOCATION.
001770    WRITE PRINT-LINE FROM HDG-LINE-ONE
001780        AFTER ADVANCING PAGE.
001790
001800 070-WRITE-SALESMAN-TOTAL.
001810    MOVE THIS-SALESMAN-TOTAL TO PRT-SALESMAN-TOTAL.
001820    WRITE PRINT-LINE FROM SALESMAN-TOTAL-LINE
001830        AFTER ADVANCING 2 LINES.
001840
001850 075-WRITE-LOCATION-TOTAL.
001860    MOVE THIS-LOCATION-TOTAL TO PRT-LOCATION-TOTAL.
001870    WRITE PRINT-LINE FROM LOCATION-TOTAL-LINE
001880        AFTER ADVANCING 2 LINES.
001890
001900 080-WRITE-COMPANY-TOTAL.
001910    MOVE COMPANY-TOTAL TO PRT-COMPANY-TOTAL.
001920    WRITE PRINT-LINE FROM COMPANY-TOTAL-LINE
001930        AFTER ADVANCING 5 LINES.

```

FIGURE 9.8 *Continued*

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID. EDECODER.
000120 AUTHOR. R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER. TRS-80.
000170 OBJECT-COMPUTER. TRS-80.
000180
000190 DATA DIVISION.
000200 WORKING-STORAGE SECTION.
000210 01 PROGRAM-SWITCHES.
000220 05 WS-LOCATION-SWITCH PIC X(3) VALUE SPACES.
000230
000240 01 PROGRAM-SUBSCRIPTS.
000250 05 LOCATION-SUB PIC 99.
000260
000270 01 LOCATION-VALUES.
000280 05 FILLER PIC X(18) VALUE "ATLANTA" ".
000290 05 FILLER PIC X(18) VALUE "BOSTON" ".
000300 05 FILLER PIC X(18) VALUE "CHICAGO" ".
000310 05 FILLER PIC X(18) VALUE "DETROIT" ".
000320 05 FILLER PIC X(18) VALUE "LOUISVILLE" ".
000330 05 FILLER PIC X(18) VALUE "MINNEAPOLIS" ".
000340 05 FILLER PIC X(18) VALUE "NEWARK" ".
000350 05 FILLER PIC X(18) VALUE "NY NEW YORK" ".
000360 05 FILLER PIC X(18) VALUE "SA SAN ANTONIO" ".
000370 05 FILLER PIC X(18) VALUE "SF SAN FRANCISCO".
000380
000390 01 LOCATION-TABLE REDEFINES LOCATION-VALUES.
000400 05 LOCATION-CODE-AND-VALUE OCCURS 10 TIMES.
000410 10 LOCATION-CODE PIC X(3).
000420 10 EXPANDED-LOCATION PIC X(15).
000430
000440 LINKAGE SECTION.
000450 01 INCOMING-LOCATION-CODE PIC X(3).
000460 01 EXPANDED-PRINT-LOCATION PIC X(15).
000470
000480 PROCEDURE DIVISION
000490 USING INCOMING-LOCATION-CODE
000500 EXPANDED-PRINT-LOCATION.
000510
000520 400-MAINLINE.
000540 PERFORM 500-EXPAND-LOCATION-CODE
000550 VARYING LOCATION-SUB FROM 1 BY 1
000560 UNTIL WS-LOCATION-SWITCH = "YES".
000570
000580 450-RETURN-TO-MAIN.
000590 EXIT PROGRAM.
000600
000610 500-EXPAND-LOCATION-CODE.
000620 IF LOCATION-SUB > 10
000630 MOVE "UNKNOWN" TO EXPANDED-PRINT-LOCATION
000640 MOVE "YES" TO WS-LOCATION-SWITCH
000650 ELSE
000660 IF INCOMING-LOCATION-CODE = LOCATION-CODE (LOCATION-SUB)
000670 MOVE EXPANDED-LOCATION (LOCATION-SUB)
000680 TO EXPANDED-PRINT-LOCATION
000690 MOVE "YES" TO WS-LOCATION-SWITCH.
000700

```

FIGURE 9.9 Invalid subprogram

PROJECTS

Expand Project 1 from Chapter 8 to include a subprogram that accepts department code and returns a department name, which can then appear in the salary report. Use the following table of department codes:

<i>Department Code</i>	<i>Department Name</i>
100	DATA PROCESSING
150	LEGAL
200	FINANCIAL
250	MARKETING
300	MANUFACTURING
350	ACCOUNTING

IO

**TABLE
PROCESSING**

OVERVIEW The reader is assumed to be familiar with the concept of a one-level table and table lookups as presented in Chapter 6. This chapter extends that knowledge in several ways.

We begin with the basics of the OCCURS clause, then expand coverage to include indexing and variable-length tables. The procedure for table initialization and table processing is reviewed, then extended to cover dynamic loading of a table and use of SET statements. A complete program is developed to contrast various techniques for table lookups and initialization.

This material is also applicable to two-level tables. We consider the different hierarchical references possible for a two-level table and use of PERFORM VARYING to manipulate two subscripts. A second program is presented to solidify these concepts.

Finally, we consider implications for establishing three-level tables. The use of PERFORM VARYING for three subscripts is shown, as is the use of three OCCURS clauses. The author refrains, however, from presenting a complete program.

THE OCCURS CLAUSE The OCCURS clause was first introduced in Chapter 6. We now review that material and, in addition, introduce some new features associated with this statement. Syntactically, the OCCURS clause has the format:

$$\text{OCCURS} \left\{ \begin{array}{l} \text{integer-1 TIMES} \\ \text{integer-1 TO integer-2 TIMES DEPENDING ON data-name-3} \end{array} \right\}$$

[INDEXED BY index-name]

The function of the OCCURS clause is to allocate space for a table. It may appear at either the elementary (Figure 10.1a) or group level (Figure 10.1b). Figures 10.1a and 10.1b allocate the same amount of space. The difference is the way space is physically assigned as per the storage schematic.

In Figure 10.1a, LOCATION-TABLE refers collectively to the 150 positions of both tables and is not used with a subscript. Procedure Division references to either LOC-CODE or LOC-NAME, *require* a subscript to indicate the particular reference.

In Figure 10.1b, LOCATION-TABLE is defined with an OCCURS clause and consequently requires a subscript if it is referenced in the Procedure Division. LOCATION-TABLE (1), for example, refers to the 15 positions of LOC-CODE (1) and LOC-NAME (1) collectively.

As stated previously, the OCCURS clause may appear at either the elementary or group level, and neither technique offers any distinct advantage. The reader should simply choose that with which he or she is most comfortable, and which best fits the particular situation.

The OCCURS clause also makes it possible to create a *variable-length* table, and consequently *variable-length* records. Until now, all records have been fixed length; i.e., they contained a *constant* number of characters. Fixed-length records, however, frequently result in wasted space on the storage medium. Consider:

```
05 CHECKS-WRITTEN OCCURS 1 TO 100 TIMES
   DEPENDING ON NUMBER-OF-CHECKS.
10 CHECK-NUMBER      PIC 9(3).
10 CHECK-AMOUNT      PIC 9(5).
```

COBOL Code:

```

05 LOCATION-TABLE.
   10 LOC-CODE OCCURS 10 TIMES    PIC X(3).
   10 LOC-NAME OCCURS 10 TIMES    PIC X(12).

```

Storage Allocation:

LOCATION-TABLE																													
CODE (1)	CODE (2)	...	CODE (10)	NAME (1)								NAME (2)								...	NAME (10)								

FIGURE 10.1a With elementary item

COBOL Code:

```

05 LOCATION-TABLE OCCURS 10 TIMES.
   10 LOC-CODE    PIC X(3).
   10 LOC-NAME    PIC X(12).

```

Storage Allocation:

LOCATION-TABLE																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																													
LOCATION-TABLE (1)										LOCATION-TABLE (2)											LOCATION-TABLE (10)																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																								
CODE (1)	NAME (1)									CODE (2)	NAME (2)									...	CODE (10)	NAME (10)																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																							
																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																													</

FIGURE 10.1b With group item

FIGURE 10.1 The OCCURS clause

In this example, each incoming record contains from 1 to 100 checks, depending on the data-name *NUMBER-OF-CHECKS* which is a *separate* field contained elsewhere in the record. Since each check requires eight storage positions (three for the number and five for the amount), the actual space required in each record varies from 8 to 800 positions. If *fixed-length* records are used, then *every* record in the file requires the maximum space; i.e., 800 characters, even though much of that is not used. Variable-length tables provide for more *efficient* use of space by *allocating only as much as necessary in individual records*, and should be used where appropriate.

The OCCURS clause also provides for an *optional* index, in that the INDEXED BY clause is enclosed in brackets. An *index* is like a subscript, except that it is defined *with* a table rather than as a separate entry in Working-Storage. An index does not provide any additional logic capability over a subscript. It does, however, result in more efficient machine language and consequently is preferred by some programmers.

If indexes are used, they must be unique for every table; i.e., the index defined with table-1 *cannot* be used to manipulate table-2. Further, indexes cannot be initialized or incremented with a MOVE or ADD statement; rather they require a new verb, SET, which is illustrated later in the chapter. (Indexes are required if the programmer utilizes the COBOL SEARCH or SEARCH ALL; statements which facilitate implementation of a table lookup procedure. These statements, however, are *not* implemented under the current release of TRS-80 COBOL, Version 1.3B, and consequently are not discussed further.)

TABLE LOOKUPS—A REVIEW

A common use of the OCCURS clause is for “table lookups”; i.e., conversion of an incoming code to an expanded value. This procedure was first presented in Chapter 6, and is reviewed in Figure 10.2.

The Data Division entries of Figure 10.2 initialize a table of 10 location codes (3 characters each) and the corresponding expanded names (15 characters). Recall from Chapter 6 that the same COBOL entry *cannot* contain both an OCCURS and a VALUE clause, and hence the need for the REDEFINES statement. This in turn assigns a new name to previously allocated space, links the OCCURS and VALUE clauses together, and makes the subsequent table lookup possible.

The table lookup itself is straightforward. The VARYING clause of the

```

01 LOCATION-VALUES.
   05 FILLER          PIC X(18) VALUE "ATLANTA"
   05 FILLER          PIC X(18) VALUE "BOSTON"
   05 FILLER          PIC X(18) VALUE "CHICAGO"
   05 FILLER          PIC X(18) VALUE "DETROIT"
   05 FILLER          PIC X(18) VALUE "LOUISVILLE"
   05 FILLER          PIC X(18) VALUE "MINNEAPOLIS"
   05 FILLER          PIC X(18) VALUE "NEWARK"
   05 FILLER          PIC X(18) VALUE "NY NEW YORK"
   05 FILLER          PIC X(18) VALUE "SA SAN ANTONIO"
   05 FILLER          PIC X(18) VALUE "SF SAN FRANCISCO"

01 LOCATION-TABLE REDEFINES LOCATION-VALUES.
   05 LOCATION-CODE-AND-VALUE OCCURS 10 TIMES.
       10 LOCATION-CODE      PIC X(3).
       10 EXPANDED-LOCATION PIC X(15).

.
.
PROCEDURE DIVISION.
010-PREPARE-REPORT.
   MOVE "NO" TO FOUND-LOCATION-SWITCH.
   PERFORM 050-SEARCH-LOCATION-TABLE
       VARYING LOCATION-SUB FROM 1 BY 1
       UNTIL FOUND-LOCATION-SWITCH = "YES".

.
.
050-SEARCH-LOCATION-TABLE.
   IF LOCATION-SUB > 10
       MOVE "UNKNOWN" TO PRT-LOCATION
       MOVE "YES" TO FOUND-LOCATION-SWITCH
   ELSE
       IF LOCATION-CODE (LOCATION-SUB) = EMP-LOC-CODE
           MOVE EXPANDED-LOCATION (LOCATION-SUB) TO PRT-LOCATION
           MOVE "YES" TO FOUND-LOCATION-SWITCH

```

Successive VALUE clauses initialize memory locations

Gives another name to previously allocated space

Subscript is automatically initialized and incremented

Terminates search for an invalid code

Terminates search when a match is found

FIGURE 10.2 Table lookup with PERFORM VARYING

PERFORM statement automatically initializes and increments the value of LOCATION-SUB. (The original code in Figure 6.9 used a MOVE and ADD statement to initialize and increment, respectively.) Note well there are *two* conditions for terminating the table lookup; i.e., either an unknown code or a match on EMP-LOC-CODE. Finally, observe how FOUND-LOCATION-SWITCH is set to "NO" immediately prior to the PERFORM VARYING statement.

INITIALIZING TABLES DYNAMICALLY

The table of Figure 10.2 is "hard coded" into the program, a less than optimal technique. Assume, for example, that the location table is used by three other programs and that the table changes. It is then necessary to alter all four programs which reference the table; a time-consuming and error-prone procedure.

A superior method would be to initialize the location table via a COPY clause. This eliminates the need for multiple changes in that only the COPY clause is altered. Even this technique is not without problems because all four programs would have to be *recompiled*, even if they were not altered explicitly. (Remember, the COPY clause brings in elements during *compilation* rather than execution.) The ideal technique, therefore, is to initialize a table dynamically, by reading values from a file when the program is executed. This is shown in Figure 10.3.

```

01  TITLE-TABLE.
05  TITLES OCCURS 1 TO 100 TIMES
    DEPENDING ON NUMBER-OF-TITLES
    INDEXED BY TITLE-INDEX.
10  TITLE-CODE          PIC 9(3).
10  TITLE-VALUE         PIC X(17).
.
.
PROCEDURE DIVISION.
.
.
020-INITIALIZE-TITLE-TABLE.
    OPEN INPUT TITLE-FILE.
    READ TITLE-FILE
    AT END MOVE "YES" TO TITLE-FILE-SWITCH.
    PERFORM 030-READ-TITLE-FILE
        VARYING TITLE-SUB FROM 1 BY 1
        UNTIL TITLE-FILE-SWITCH = "YES".
    CLOSE TITLE-FILE.
.
.
030-READ-TITLE-FILE.
    IF TITLE-SUB > 100
        MOVE "YES" TO TITLE-FILE-SWITCH
        DISPLAY "ERROR - TITLE TABLE EXCEEDED"
    ELSE
        ADD 1 TO NUMBER-OF-TITLES
        MOVE IN-TITLE-CODE TO TITLE-CODE (TITLE-SUB)
        MOVE IN-TITLE-VALUE TO TITLE-VALUE (TITLE-SUB).
    READ TITLE-FILE
    AT END MOVE "YES" TO TITLE-FILE-SWITCH.

```

Variable length table

Initial read for title file

Checks that table size is not exceeded

Increments number of table entries

Moves value from incoming file to table

FIGURE 10.3 Initializing a table by reading from a file

The Data Division entries of Figure 10.3 contain an OCCURS clause which allocates space for a variable-length table. There are no VALUE clauses, however, because the table will be filled from an external file.

The Procedure Division opens TITLE-FILE, reads it once, and then performs the paragraph 030-READ-TITLE-FILE until it is empty. This paragraph first checks that the table size is not exceeded, then moves the title code and value just read to the current position in the table. Observe that TITLE-SUB is incremented automatically in the PERFORM VARYING statement, but that NUMBER-OF-TITLES is incremented explicitly. Finally, note both the initial read for TITLE-FILE, and that the last statement of the performed routine is a second read. This is consistent with the structured programming syntax we have followed throughout.

THE SET STATEMENT

Figure 10.4 illustrates an alternative way of implementing a table lookup; specifically it uses *indexes* rather than *subscripts*. An index is very much like a subscript in that it permits one to "step through a table." The difference is in the generated machine language—indexing provides more efficient code than subscripts; indexes do not, however, provide any additional logic capability and in that sense are redundant with subscripts.

If indexes are used, they must be defined *with* the table they reference (see Figure 10.4), rather than in Working-Storage. Further, every table requires its own unique index, whereas a single subscript can be used with several different tables. (The latter, however, is not desirable from a documentation viewpoint, and hence is not encouraged as was explained in Chapter 7.)

```

01  TITLE-TABLE.
05  TITLES OCCURS 1 TO 100 TIMES
    DEPENDING ON NUMBER-OF-TITLES
    INDEXED BY TITLE-INDEX.
10  TITLE-CODE      PIC 9(3).
10  TITLE-VALUE     PIC X(17).
.
.
.
PROCEDURE DIVISION.
.
.
.
    MOVE "NO" TO FOUND-TITLE-SWITCH.
    SET TITLE-INDEX TO 1.
    PERFORM 060-SEARCH-TITLE-TABLE
        UNTIL FOUND-TITLE-SWITCH = "YES".
.
.
.
060-SEARCH-TITLE-TABLE.
    IF TITLE-INDEX > NUMBER-OF-TITLES
        MOVE "UNKNOWN" TO PRT-TITLE
        MOVE "YES" TO FOUND-TITLE-SWITCH
    ELSE
        IF TITLE-CODE (TITLE-INDEX) = EMP-TITLE-CODE
            MOVE TITLE-VALUE (TITLE-INDEX) TO PRT-TITLE
            MOVE "YES" TO FOUND-TITLE-SWITCH
        ELSE
            SET TITLE-INDEX UP BY 1.

```

Index is defined with table

Initializes index

Increments index

FIGURE 10.4 Table lookup with SET statements

Indexes are manipulated in a SET statement, and may *not* be referenced in either a MOVE or an ADD. Compare the table lookup of Figure 10.4 with the code of Figure 6.9. The latter initialized a subscript, WS-MAJOR-SUB, with a MOVE statement and subsequently incremented the subscript with an ADD statement. These operations are accomplished in Figure 10.4 by the statements:

```
SET TITLE-INDEX TO 1
```

and

```
SET TITLE-INDEX UP BY 1
```

to initialize and increment, respectively. The remainder of Figure 10.4 parallels the logic of Figure 6.9, and should pose no difficulty.

One last point—*although a table is defined with an index, it may still be referenced with a subscript*. The code in Figure 10.3, for example, used a subscript, TITLE-SUB, to reference the TITLES table, even though the latter was defined with an index. The converse is not true, however. That is, one may not use indexes to process a table unless it has first been defined with an index.

A COMPLETE EXAMPLE Information on table processing is summarized via the COBOL program of Figure 10.5. Processing specifications are as follows:

<i>Input:</i>	A file of employee records, with each record containing a code for the employee's title, location, education, and performance. Test data are shown in Figure 10.6a.
<i>Processing:</i>	Initialize the table of title codes and expanded values by reading from the file of Figure 10.6b. Process the file of incoming employee records to produce a set of "personnel profiles" with expanded information for each employee.
<i>Output:</i>	The set of employee profiles as described above. The printed output should also contain the employee's percent salary increase (where appropriate) and the elapsed time (in months) between increases. Partial output from the program is shown in Figure 10.6c.

Figure 10.5 contains the completed program for table processing. The incoming employee file is defined in lines 380–520. The employee record contains a one-level table for salary data, and two occurrences are stored. EMP-SALARY (1) refers to the employee's present salary, whereas EMP-SALARY (2) is the previous salary. These data names are used in the COMPUTE statement of lines 2920 and 2930, and the reader is urged to verify

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID. TABLES.
000120 AUTHOR. R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER. TRS-80.
000170 OBJECT-COMPUTER. TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210     SELECT TITLE-FILE
000220         ASSIGN TO INPUT "TITLES/DAT".
000230     SELECT EMPLOYEE-FILE
000240         ASSIGN TO INPUT "TABLES/DAT".
000250     SELECT PRINT-FILE
000260         ASSIGN TO PRINT "TABLES/TXT".
000270
000280 DATA DIVISION.
000290 FILE SECTION.
000300 FD TITLE-FILE
000310     LABEL RECORDS ARE STANDARD
000320     RECORD CONTAINS 20 CHARACTERS
000330     DATA RECORD IS SALES-RECORD.
000340 01 TITLE-RECORD.
000350     05 IN-TITLE-CODE PIC 9(3).
000360     05 IN-TITLE-VALUE PIC X(17).
000370
000380 FD EMPLOYEE-FILE
000390     LABEL RECORDS ARE STANDARD
000400     RECORD CONTAINS 50 CHARACTERS
000410     DATA RECORD IS EMPLOYEE-RECORD.
000420 01 EMPLOYEE-RECORD.
000430     05 EMP-SOC-SEC-NUMBER PIC 9(9).
000440     05 EMP-LAST-NAME PIC X(15).
000450     05 EMP-LOC-CODE PIC X(3).
000460     05 EMP-TITLE-CODE PIC 999.
000470     05 EMP-EDUCATION-CODE PIC 9.
000480     05 EMP-PERFORMANCE-CODE PIC X.
000490     05 EMP-SALARY-DATA OCCURS 2 TIMES.
000500         10 EMP-SALARY PIC 9(5).
000510         10 EMP-SALARY-MONTH PIC 99.
000520         10 EMP-SALARY-YEAR PIC 99.
000530
000540 FD PRINT-FILE
000550     LABEL RECORDS ARE STANDARD
000560     RECORD CONTAINS 132 CHARACTERS
000570     DATA RECORD IS PRINT-LINE.
000580 01 PRINT-LINE PIC X(132).
000590
000600 WORKING-STORAGE SECTION.
000610 01 NUMBER-OF-TITLES PIC 9(3) VALUE ZEROS.
000620
000630 01 PROGRAM-SWITCHES.
000640     05 EMPLOYEE-FILE-SWITCH PIC X(3) VALUE SPACES.
000650     05 TITLE-FILE-SWITCH PIC X(3) VALUE SPACES.
000660     05 FOUND-LOCATION-SWITCH PIC X(3) VALUE SPACES.
000670     05 FOUND-TITLE-SWITCH PIC X(3) VALUE SPACES.
000680
000690 01 PROGRAM-SUBSCRIPTS.
000700     05 LOCATION-SUB PIC 99.
000710     05 TITLE-SUB PIC 99.
000720
000730 01 SALARY-CALCULATIONS.
000740     05 PERCENT-SALARY-INCREASE PIC 99V9.
000750     05 MONTHS-BETWEEN-INCREASE PIC 99.
000760
000770 01 LOCATION-VALUES.
000780     05 FILLER PIC X(18) VALUE "ATLANTA" ".
000790     05 FILLER PIC X(18) VALUE "BOSTON" ".
000800     05 FILLER PIC X(18) VALUE "CHICAGO" ".
000810     05 FILLER PIC X(18) VALUE "DETROIT" ".

```

Contains the file of title codes for table initialization

Table for employee's salary data

Associated with title table (See line 980)

Each table has its own subscript

FIGURE 10.5 Table processing

```

000820 05 FILLER PIC X(18) VALUE "LOULOUISVILLE ".
000830 05 FILLER PIC X(18) VALUE "MINMINNEAPOLIS ".
000840 05 FILLER PIC X(18) VALUE "NEWNEWARK ".
000850 05 FILLER PIC X(18) VALUE "NY NEW YORK ".
000860 05 FILLER PIC X(18) VALUE "SA SAN ANTONIO ".
000870 05 FILLER PIC X(18) VALUE "SF SAN FRANCISCO".
000880
000890 01 LOCATION-TABLE REDEFINES LOCATION-VALUES.
000900 05 LOCATION-CODE-AND-VALUE OCCURS 10 TIMES.
000910 10 LOCATION-CODE PIC X(3).
000920 10 EXPANDED-LOCATION PIC X(15).
000930
000940 COPY "EDUCATE/CBL". — Copy initializes table
000001 01 EDUCATION-TABLE.
000002 05 EDUCATION-VALUES.
000003 10 FILLER PIC X(10) VALUE "SOME H.S. ".
000004 10 FILLER PIC X(10) VALUE "HS DIPLOMA".
000005 10 FILLER PIC X(10) VALUE "2YR DEGREE".
000006 10 FILLER PIC X(10) VALUE "4YR DEGREE".
000007 10 FILLER PIC X(10) VALUE "SOME GRAD ".
000008 10 FILLER PIC X(10) VALUE "MASTERS ".
000009 10 FILLER PIC X(10) VALUE "PH. D. ".
000010 10 FILLER PIC X(10) VALUE "OTHER ".
000011
000012 05 EDU-NAME REDEFINES EDUCATION-VALUES
000013 OCCURS 8 TIMES PIC X(10).
000014
000950
000960 01 TITLE-TABLE.
000970 05 TITLES OCCURS 1 TO 100 TIMES
000980 DEPENDING ON NUMBER-OF-TITLES
000990 INDEXED BY TITLE-INDEX. — No further definition is required
001000 10 TITLE-CODE PIC 9(3).
001010 10 TITLE-VALUE PIC X(17).
001020
001030 01 PRINT-LINE-ONE.
001040 05 FILLER PIC X(22) VALUE SPACES.
001050 05 FILLER PIC X(17)
001060 VALUE "PERSONNEL PROFILE".
001070 05 FILLER PIC X(93) VALUE SPACES.
001080
001090 01 PRINT-LINE-TWO.
001100 05 FILLER PIC X(6) VALUE "NAME: ".
001110 05 PRT-LAST-NAME PIC X(15).
001120 05 FILLER PIC X(25) VALUE SPACES.
001130 05 FILLER PIC X(13) VALUE "SOC SEC NUM: ".
001140 05 PRT-SOC-SEC-NUMBER PIC 999B99B9999.
001150 05 FILLER PIC X(62) VALUE SPACES.
001160
001170 01 PRINT-LINE-THREE.
001180 05 FILLER PIC X(10) VALUE "LOCATION: ".
001190 05 PRT-LOCATION PIC X(15).
001200 05 FILLER PIC X(21) VALUE SPACES.
001210 05 FILLER PIC X(7) VALUE "TITLE: ".
001220 05 PRT-TITLE PIC X(17).
001230 05 FILLER PIC X(62) VALUE SPACES.
001240
001250 01 PRINT-LINE-FOUR.
001260 05 FILLER PIC X(11) VALUE "EDUCATION: ".
001270 05 PRT-EDUCATION PIC X(10).
001280 05 FILLER PIC X(25) VALUE SPACES.
001290 05 FILLER PIC X(13) VALUE "PERFORMANCE: ".
001300 05 PRT-PERFORMANCE PIC X(11).
001310 05 FILLER PIC X(62) VALUE SPACES.
001320
001330 01 PRINT-LINE-FIVE.
001340 05 FILLER PIC X(25) VALUE SPACES.
001350 05 FILLER PIC X(11) VALUE "SALARY DATA".
001360 05 FILLER PIC X(96) VALUE SPACES.
001370
001380 01 PRINT-LINE-SIX.
001390 05 FILLER PIC X(5) VALUE SPACES.

```

FIGURE 10.5 Continued

```

001400      05 FILLER                      PIC X(6)  VALUE "SALARY".
001410      05 FILLER                      PIC X(10) VALUE SPACES.
001420      05 FILLER                      PIC X(4)  VALUE "DATE".
001430      05 FILLER                      PIC X(10) VALUE SPACES.
001440      05 FILLER                      PIC X(10) VALUE "% INCREASE".
001450      05 FILLER                      PIC X(10) VALUE SPACES.
001460      05 FILLER                      PIC X(3)  VALUE "MBI".
001470      05 FILLER                      PIC X(74) VALUE SPACES.
001480
001490 01 PRINT-SALARY-LINE.
001500      05 FILLER                      PIC X(4)  VALUE SPACES.
001510      05 PRT-SALARY                  PIC $99,999.
001520      05 FILLER                      PIC X(9)  VALUE SPACES.
001530      05 PRT-SALARY-MONTH          PIC Z9.
001540      05 PRT-SLASH                  PIC X      VALUE "/".
001550      05 PRT-SALARY-YEAR            PIC Z9.
001560      05 FILLER                      PIC X(12) VALUE SPACES.
001570      05 PRT-SALARY-INCREASE        PIC Z9.99.
001580      05 FILLER                      PIC X(13) VALUE SPACES.
001590      05 PRT-SALARY-MBI             PIC ZZ.
001600      05 FILLER                      PIC X(75) VALUE SPACES.
001610
001620 PROCEDURE DIVISION.
001630 010-PREPARE-REPORT.
001640      PERFORM 020-INITIALIZE-TITLE-TABLE.
001650
001660      OPEN INPUT EMPLOYEE-FILE
001670      OUTPUT PRINT-FILE.
001680      READ EMPLOYEE-FILE
001690      AT END MOVE "YES" TO EMPLOYEE-FILE-SWITCH.
001700      PERFORM 040-WRITE-PERSONNEL-PROFILES
001710      UNTIL EMPLOYEE-FILE-SWITCH = "YES".
001720      CLOSE EMPLOYEE-FILE
001730      PRINT-FILE.
001740      STOP RUN.
001750
001760 020-INITIALIZE-TITLE-TABLE.
001770      OPEN INPUT TITLE-FILE.
001780      READ TITLE-FILE
001790      AT END MOVE "YES" TO TITLE-FILE-SWITCH.
001800      PERFORM 030-READ-TITLE-FILE
001810      VARYING TITLE-SUB FROM 1 BY 1
001820      UNTIL TITLE-FILE-SWITCH = "YES".
001830      CLOSE TITLE-FILE.
001840
001850 030-READ-TITLE-FILE.
001860      IF TITLE-SUB > 100
001870      MOVE "YES" TO TITLE-FILE-SWITCH
001880      DISPLAY "ERROR - TITLE TABLE EXCEEDED"
001890      ELSE
001900      ADD 1 TO NUMBER-OF-TITLES
001910      MOVE IN-TITLE-CODE TO TITLE-CODE (TITLE-SUB)
001920      MOVE IN-TITLE-VALUE TO TITLE-VALUE (TITLE-SUB).
001930      READ TITLE-FILE
001940      AT END MOVE "YES" TO TITLE-FILE-SWITCH.
001950
001960 040-WRITE-PERSONNEL-PROFILES.
001970      WRITE PRINT-LINE FROM PRINT-LINE-ONE
001980      AFTER ADVANCING PAGE.
001990
002000      MOVE EMP-LAST-NAME TO PRT-LAST-NAME.
002010      MOVE EMP-SOC-SEC-NUMBER TO PRT-SOC-SEC-NUMBER.
002020      INSPECT PRT-SOC-SEC-NUMBER
002030      REPLACING ALL " " BY "-".
002040      WRITE PRINT-LINE FROM PRINT-LINE-TWO
002050      AFTER ADVANCING 2 LINES.
002060
002070      MOVE "NO" TO FOUND-LOCATION-SWITCH.
002080      PERFORM 050-SEARCH-LOCATION-TABLE
002090      VARYING LOCATION-SUB FROM 1 BY 1
002100      UNTIL FOUND-LOCATION-SWITCH = "YES".

```

Initial read of employee file

Explained in detail - see Figure 10.3

Edits social security number

Initiates location search

FIGURE 10.5 Continued

```

002110
002120     MOVE "NO" TO FOUND-TITLE-SWITCH.
002130     SET TITLE-INDEX TO 1.
002140     PERFORM 060-SEARCH-TITLE-TABLE
002150         UNTIL FOUND-TITLE-SWITCH = "YES".
002160
002170     WRITE PRINT-LINE FROM PRINT-LINE-THREE
002180         AFTER ADVANCING 2 LINES.
002190
002200     PERFORM 070-EXPAND-EDUCATION-CODE.
002210     PERFORM 080-EXPAND-PERFORMANCE-CODE.
002220     WRITE PRINT-LINE FROM PRINT-LINE-FOUR
002230         AFTER ADVANCING 2 LINES.
002240
002250     WRITE PRINT-LINE FROM PRINT-LINE-FIVE
002260         AFTER ADVANCING 5 LINES.
002270
002280     WRITE PRINT-LINE FROM PRINT-LINE-SIX
002290         AFTER ADVANCING 2 LINES.
002300
002310     MOVE SPACES TO PRINT-SALARY-LINE.
002320     MOVE "/" TO PRT-SLASH.
002330     MOVE EMP-SALARY (1) TO PRT-SALARY.
002340     MOVE EMP-SALARY-MONTH (1) TO PRT-SALARY-MONTH.
002350     MOVE EMP-SALARY-YEAR (1) TO PRT-SALARY-YEAR.
002360
002370     IF EMP-SALARY (2) > 0
002380         PERFORM 090-EVALUATE-SALARY-INCREASES
002390         MOVE PERCENT-SALARY-INCREASE TO PRT-SALARY-INCREASE
002400         MOVE MONTHS-BETWEEN-INCREASE TO PRT-SALARY-MBI
002410         WRITE PRINT-LINE FROM PRINT-SALARY-LINE
002420             AFTER ADVANCING 2 LINES
002430         MOVE SPACES TO PRINT-SALARY-LINE
002440         MOVE "/" TO PRT-SLASH
002450         MOVE EMP-SALARY (2) TO PRT-SALARY
002460         MOVE EMP-SALARY-MONTH (2) TO PRT-SALARY-MONTH
002470         MOVE EMP-SALARY-YEAR (2) TO PRT-SALARY-YEAR.
002480     WRITE PRINT-LINE FROM PRINT-SALARY-LINE
002490         AFTER ADVANCING 1 LINE.
002500     READ EMPLOYEE-FILE
002510         AT END MOVE "YES" TO EMPLOYEE-FILE-SWITCH.
002520
002530 050-SEARCH-LOCATION-TABLE.
002540     IF LOCATION-SUB > 10
002550         MOVE "UNKNOWN" TO PRT-LOCATION
002560         MOVE "YES" TO FOUND-LOCATION-SWITCH
002570     ELSE
002580         IF LOCATION-CODE (LOCATION-SUB) = EMP-LOC-CODE
002590             MOVE EXPANDED-LOCATION (LOCATION-SUB) TO PRT-LOCATION
002600             MOVE "YES" TO FOUND-LOCATION-SWITCH.
002610
002620 060-SEARCH-TITLE-TABLE.
002630     IF TITLE-INDEX > NUMBER-OF-TITLES
002640         MOVE "UNKNOWN" TO PRT-TITLE
002650         MOVE "YES" TO FOUND-TITLE-SWITCH
002660     ELSE
002670         IF TITLE-CODE (TITLE-INDEX) = EMP-TITLE-CODE
002680             MOVE TITLE-VALUE (TITLE-INDEX) TO PRT-TITLE
002690             MOVE "YES" TO FOUND-TITLE-SWITCH
002700         ELSE
002710             SET TITLE-INDEX UP BY 1.
002720
002730 070-EXPAND-EDUCATION-CODE.
002740     IF EMP-EDUCATION-CODE < 1 OR EMP-EDUCATION-CODE > 8
002750         MOVE "UNKNOWN" TO PRT-EDUCATION
002760     ELSE
002770         MOVE EDU-NAME (EMP-EDUCATION-CODE) TO PRT-EDUCATION.
002780
002790 080-EXPAND-PERFORMANCE-CODE.

```

Avoids division by zero

Explained in detail - see Figure 10.2

Checks for invalid location code

Direct access to table entries

FIGURE 10.5 Continued


```

002800 IF EMP-PERFORMANCE-CODE = "E"
002810     MOVE "EXCELLENT" TO PRT-PERFORMANCE
002820 ELSE
002830     IF EMP-PERFORMANCE-CODE = "A"
002840         MOVE "AVERAGE" TO PRT-PERFORMANCE
002850     ELSE
002860         IF EMP-PERFORMANCE-CODE = "P"
002870             MOVE "POOR" TO PRT-PERFORMANCE
002880         ELSE
002890             MOVE "UNKNOWN" TO PRT-PERFORMANCE.
002900
002910 090-EVALUATE-SALARY-INCREASES.
002920     COMPUTE PERCENT-SALARY-INCREASE
002930         = 100 * (EMP-SALARY (1) - EMP-SALARY (2)) / EMP-SALARY (2).
002940
002950     COMPUTE MONTHS-BETWEEN-INCREASE
002960         = (EMP-SALARY-YEAR (1) - EMP-SALARY-YEAR (2)) * 12
002970         + (EMP-SALARY-MONTH (1) - EMP-SALARY-MONTH (2)).

```

Expands performance code

Previous salary

Present salary

FIGURE 10.5 Continued

the correctness of this calculation. (This is best accomplished by “plugging numbers in” and “playing computer.” Present and previous salaries of \$12,000 and \$10,000, for example, should yield a percent increase of 20%.) In similar fashion, lines 2950-2970 calculate the months between increase, and the reader should verify this statement as well. Observe also the check in line 2370 to verify that a second level of salary is in fact present. Failure to do this would result in an attempted division by zero.

The tables for location, title, and education are each initialized differently. The location table, lines 770-920 is “hard coded” in the program. It is accessed in the PERFORM/VARYING of lines 2080-2100 as was previously explained in the discussion associated with Figure 10.2.

The education table is initialized through the COPY statement of line 940. (Recall from Chapter 9 that lines 1-14, which appear after the COPY, are brought in from the external file EDUCATE/CBL during compilation.) The education code is expanded in lines 2730-2770 via “direct access” to the table, rather than a sequential search. The education codes themselves are numeric, and consecutive (from 1 to 8), but are *not* stored in the table, because *the position within the table corresponds to the code*. Hence, “SOME HS” has a code of 1, “HS DIPLOMA” a code of 2, and so on. (See the table definition in the Data Division.) The table lookup is a one-line procedure (line 2770) which moves the table entry in the desired position to the print line; e.g. an education code of 6 will move MASTERS to the print line. Observe also the check for an invalid code in line 2740.

Space for the title table is allocated in lines 960-1010. It is initialized by reading values from a file (lines 1760-1940) as explained earlier in conjunction with Figure 10.3. The actual table lookup (lines 2620-2710) uses indexes and the SET statement as per the discussion of Figure 10.4.

Employee performance is expanded in the nested IF of lines 2800-2890. The performance table is implicitly defined within this statement.

The Data Division requires the definition of several print lines (lines 1030-1600). The Procedure Division opens the files, does an initial read for the employee file, then executes 040-WRITE-PERSONNEL-PROFILES until there are no more employee records. This routine in turn calls several lower level modules to implement the various table lookups and prints the profiles.

11111111SMITH	ATL0105A180000781000000000	Title code
22222222JONES	CHI0404E230000381200000980	
33333333BAKER	BOS0306P195000182180001280	
44444444MILGROM	DET0207G280000481240000480	
55555555CRAWFORD	WAS0305E300000581250001179	

a - Employee File

010ACCOUNTANT	Title code and expanded value
020AUDITOR	
030MANAGER	
040PROGRAMMER	
050WALL STREET REP	

b - Title File

PERSONNEL PROFILE

NAME: JONES	SOC SEC NUM: 222-22-2222
LOCATION: CHICAGO	TITLE: PROGRAMMER
EDUCATION: 4YR DEGREE	PERFORMANCE: EXCELLENT

SALARY DATA

SALARY	DATE	% INCREASE	MBI
\$23,000	3/81	15.00	6
\$20,000	9/80		

PERSONNEL PROFILE

NAME: SMITH	SOC SEC NUM: 111-11-1111
LOCATION: ATLANTA	TITLE: ACCOUNTANT
EDUCATION: SOME GRAD	PERFORMANCE: AVERAGE

SALARY DATA

SALARY	DATE	% INCREASE	MBI
\$18,000	7/81		

c - Partial Output (First two Employee Records)

FIGURE 10.6 Test data and partial output for table processing program. (a) Employee file. (b) Title file. (c) Partial output (first two employee records).

TWO-LEVEL TABLES Two-dimension tables require two subscripts to specify a particular entry. Consider Figure 10.7 which shows a two-dimension table to determine entry-level salaries in Company X. Personnel has established a policy that starting salary is a function of both responsibility level (values 1 to 10) and experience (values 1 to 5). Thus an employee with responsibility level of 4 and experience level of 1 would receive \$10,000. An employee with responsibility of 1 and experience of 4 would receive \$9,000.

$$\begin{array}{l} \left\{ \begin{array}{l} \text{Responsibility level} = 4 \\ \text{Experience level} = 1 \end{array} \right. \end{array} \quad \begin{array}{l} \left\{ \begin{array}{l} \text{Responsibility level} = 1 \\ \text{Experience level} = 4 \end{array} \right. \end{array}$$

FIGURE 10.7 Entry-level salary (illustration of two-dimension tables)

Establishment of space for this table in COBOL requires Data Division entries as follows:

```

01 SALARY-TABLE.
   05 SALARY-RESPONSIBILITY OCCURS 10 TIMES.
      10 SALARY-EXPERIENCE OCCURS 5 TIMES          PIC 9(5).

```

These entries would cause a total of 250 consecutive storage positions to be allocated ($10 \times 5 \times 5$) as shown:

[illegible]

Each of the 10 salary responsibility levels has 5 experience levels associated with it. Once we realize that the first 25 storage positions refer to the first responsibility level, the next 25 to the second responsibility level, etc., we know how to initialize the table using the `VALUE` and `REDEFINES` clauses. Note that the level number for experience (10) is higher than for responsibility (05), indicating that experience belongs to responsibility; i.e., responsibility is a *group* item, whereas experience is an *elementary* item. Indeed, if the level numbers were the same, `SALARY-TABLE` would not be two-dimensional.

Any Procedure Division reference to `SALARY-EXPERIENCE` requires *two* subscripts, which appear in the *same order as the OCCURS clauses*. The first subscript refers to responsibility, and the second one to experience. `SALARY-EXPERIENCE (10, 5)` is a valid reference, but `SALARY-EXPERIENCE (5, 10)` is invalid. The former denotes responsibility and experience levels of 10 and 5, respectively. However, the latter denotes a responsibility level of 5 and an experience level of 10, which do not exist.

PERFORM VARYING

The `PERFORM VARYING` statement is a convenient way to manipulate subscripts in either one- or two-level tables. Consider the syntactical format:

```
PERFORM procedure-name-1 [THRU procedure-name-2]
```

```
VARYING identifier-1 FROM {identifier-2  
literal-2 }
```

```
BY {identifier-3  
literal-3 } UNTIL condition-1
```

```
[AFTER identifier-4 FROM {identifier-5  
literal-5 }]
```

```
BY {identifier-6  
literal-6 } UNTIL condition-2]
```

and an example:

```
PERFORM INITIALIZE-TOTALS  
  VARYING QUESTION-SUB FROM 1 BY 1  
    UNTIL QUESTION-SUB > 10  
  AFTER ANSWER-SUB FROM 1 BY 1  
    UNTIL ANSWER-SUB > 3.
```

The analysis of the statement is facilitated by first reviewing `PERFORM VARYING` with a single subscript. Recall that the statement

```
PERFORM PAR-A  
  VARYING SUBSCRIPT FROM 1 BY 1  
    UNTIL SUBSCRIPT = 4.
```

will perform PAR-A three times rather than four. This is because the `VARYING/UNTIL` combination *increments, tests, and then branches*. Accordingly, after `PAR-A` is executed three times, `SUBSCRIPT` is incremented to 4. The condition `UNTIL SUBSCRIPT = 4` is satisfied and `PAR-A` will not be performed again. In other words, if a paragraph is to be executed *N* times, then

the condition must specify `UNTIL SUBSCRIPT > N`, rather than `SUBSCRIPT = N`.

Extending this logic to the two subscripts of the preceding example, we conclude that the procedure `INITIALIZE-TOTALS` will be executed 30 (10×3) times. (Note well the greater-than signs in the `PERFORM` statement.) The remaining concern is in which order will the subscripts, `QUESTION-SUB` and `ANSWER-SUB`, be varied. The answer is that the *bottom* subscript (i.e., the subscript in the `AFTER` clause) is varied first.

The first time `INITIALIZE-TOTALS` is executed, `QUESTION-SUB` and `ANSWER-SUB` are both equal to 1. The second and third times `QUESTION-SUB` is held at 1 while `ANSWER-SUB` is incremented to 2, then 3. After `INITIALIZE-TOTALS` has been performed three times, the `AFTER` condition is satisfied. Hence, the fourth time out `QUESTION-SUB` is incremented to 2, and `ANSWER-SUB` is *reset* to 1. The situation is more clearly explained by the table of Figure 10.8.

	QUESTION-SUB	ANSWER-SUB
1st execution	1	1
2nd execution	1	2
3rd execution	1	3
4th execution	2	1
5th execution	2	2
6th execution	2	3
7th execution	3	1
...
28th execution	10	1
29th execution	10	2
30th execution	10	3

FIGURE 10.8 `PERFORM VARYING` with two subscripts

A Complete Example

We further develop use of two-dimension tables by considering requirements for another COBOL program. Let us assume that a survey of 10 questions has been distributed and returned. Every question in the survey has 3 possible answers; yes, no, and not sure (denoted by Y, N, or X, respectively). The answers to all questions on a given survey are entered in positions 1-10 of a single record. There are as many records as there are survey respondents; i.e., each record contains 10 responses from one individual.

Figure 10.9 shows various ways of representing a two-dimension table. Figure 10.9a illustrates the way the survey tabulations would probably appear in a report. Note that the object of the COBOL program is to process the completed questionnaires and compute the numbers in Figure 10.9a (which contains hypothetical values for 50 surveys).

Figure 10.9b contains the COBOL entries for establishing a two-dimension table of 10 rows and 3 columns. Realize that Figure 10.9b merely allocates space but does *not* assign values to the table.

Figure 10.9c shows the storage allocation resulting from Figure 10.9b. A total of 60 storage positions are allocated, two for each of the 30 table entries. Each Procedure Division reference to `QUESTION-NUMBER` requires *one* subscript; e.g., `QUESTION-NUMBER (2)`, which refers collectively to the three answers for the second question. Each reference to `ANSWER`, however, requires *two* subscripts denoting the question number and answer.

Question No.	Number of Responses		
	Yes	No	Not Sure
1	15	20	15
2	20	18	12
3	6	23	21
4	24	21	5
5	38	11	1
6	16	32	2
7	10	20	20
8	35	10	5
9	4	39	7
10	46	3	1

— Six people responded yes to the third question, i.e. the element in row 3, column 1, has a value of 6.

(a)

```

01 SURVEY-RESPONSES.
05 QUESTION-NUMBER OCCURS 10 TIMES.
10 ANSWER OCCURS 3 TIMES PIC 99.

```

(b)

SURVEY-RESPONSES											
QUESTION-NUMBER (1)			QUESTION-NUMBER (2)			QUESTION-NUMBER (10)				
ANS (1)	ANS (2)	ANS (3)	ANS (1)	ANS (2)	ANS (3)		ANS (1)	ANS (2)	ANS (3)		
						...					

— Any reference to ANS requires 2 subscripts

(c)

— Any reference to QUESTION-NUMBER requires 1 subscript

FIGURE 10.9 Two-dimension tables. (a) Conceptual view. (b) COBOL entries (allocates space but does not assign values). (c) Storage allocation

Hence, the value contained in the table element ANSWER (4, 2) is the number of “no” answers to the fourth question.

Figure 10.10 contains the completed program for processing the surveys. Lines 620 through 640 define the two-dimension table to hold the survey responses, as explained in conjunction with Figure 10.9. Note also COBOL line 330, which defines a one-dimension table, SURVEY-RESPONSE, to reference the 10 responses for each survey.

The Procedure Division begins by opening the survey and print files. The SURVEY-RESPONSE table is initialized through a PERFORM VARYING

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID. SURVEY.
000120 AUTHOR. R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER. TRS-80.
000170 OBJECT-COMPUTER. TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210     SELECT SURVEY-FILE
000220         ASSIGN TO INPUT "SURVEY/DAT".
000230     SELECT PRINT-FILE
000240         ASSIGN TO PRINT "SURVEY/TXT".
000250
000260 DATA DIVISION.
000270 FILE SECTION.
000280 FD SURVEY-FILE
000290     LABEL RECORDS ARE STANDARD
000300     RECORD CONTAINS 10 CHARACTERS
000310     DATA RECORD IS SURVEY-RECORD.
000320 01 SURVEY-RECORD.
000330     05 SURVEY-RESPONSE OCCURS 10 TIMES PIC X.
000340
000350 FD PRINT-FILE
000360     LABEL RECORDS ARE STANDARD
000370     RECORD CONTAINS 132 CHARACTERS
000380     DATA RECORDS ARE DETAIL-LINE HEADING-LINE.
000390 01 HEADING-LINE.
000400     05 FILLER PIC X(4).
000410     05 HEADING-INFORMATION PIC X(30).
000420     05 FILLER PIC X(98).
000430
000440 01 DETAIL-LINE.
000450     05 FILLER PIC X(7).
000460     05 QUESTION PIC ZZ.
000470     05 FILLER PIC X(6).
000480     05 PRINT-YES PIC ZZ.
000490     05 FILLER PIC X(3).
000500     05 PRINT-NO PIC ZZ.
000510     05 FILLER PIC X(6).
000520     05 PRINT-NOT-SURE PIC ZZ.
000530     05 FILLER PIC X(102).
000540
000550
000560 WORKING-STORAGE SECTION.
000570 01 WS-DATA-REMAINS-SWITCH PIC X(3) VALUE SPACES.
000580 01 SURVEY-SUBSCRIPTS.
000590     05 WS-ANSWER-SUB PIC 99.
000600     05 WS-QUESTION-SUB PIC 99.
000610
000620 01 SURVEY-RESPONSES.
000630     05 QUESTION-NUMBER OCCURS 10 TIMES.
000640         10 ANSWER OCCURS 3 TIMES PIC 99.
000650
000660
000670 PROCEDURE DIVISION.
000680 000-PROCESS-SURVEYS.
000690     OPEN INPUT SURVEY-FILE
000700     OUTPUT PRINT-FILE.
000710
000720     PERFORM 005-INITIALIZE-SURVEY-TOTALS
000730     VARYING WS-QUESTION-SUB FROM 1 BY 1
000740     UNTIL WS-QUESTION-SUB > 10
000750     AFTER WS-ANSWER-SUB FROM 1 BY 1
000760     UNTIL WS-ANSWER-SUB > 3.
000770

```

Definition of One-Level Table

Definition of Two-Level Table

Performed routine is executed thirty times

FIGURE 10.10 Two-level tables

```

000780 READ SURVEY-FILE
000790 AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
000800 PERFORM 010-PROCESS-SURVEYS
000810 UNTIL WS-DATA-REMAINS-SWITCH = "NO".
000820
000830 PERFORM 018-WRITE-HEADING.
000840
000850 PERFORM 020-WRITE-RESPONSES
000860 VARYING WS-QUESTION-SUB FROM 1 BY 1
000870 UNTIL WS-QUESTION-SUB > 10.
000880
000890 CLOSE SURVEY-FILE
000900 PRINT-FILE.
000910 STOP RUN.
000920
000930 005-INITIALIZE-SURVEY-TOTALS.
000940 MOVE ZERO TO ANSWER (WS-QUESTION-SUB, WS-ANSWER-SUB).
000950
000960 010-PROCESS-SURVEYS.
000970 PERFORM 015-EVALUATE-RESPONSE
000980 VARYING WS-QUESTION-SUB FROM 1 BY 1
000990 UNTIL WS-QUESTION-SUB > 10.
001000 READ SURVEY-FILE
001010 AT END MOVE "NO" TO WS-DATA-REMAINS-SWITCH.
001020
001030 015-EVALUATE-RESPONSE.
001040 IF SURVEY-RESPONSE (WS-QUESTION-SUB) = "Y"
001050 MOVE 1 TO WS-ANSWER-SUB
001060 ELSE
001070 IF SURVEY-RESPONSE (WS-QUESTION-SUB) = "N"
001080 MOVE 2 TO WS-ANSWER-SUB
001090 ELSE
001100 MOVE 3 TO WS-ANSWER-SUB.
001110
001120 ADD 1 TO ANSWER (WS-QUESTION-SUB, WS-ANSWER-SUB).
001130
001140 018-WRITE-HEADING.
001150 MOVE SPACES TO HEADING-LINE.
001160 MOVE "QUESTION YES NO NOT SURE"
001170 TO HEADING-INFORMATION.
001180 WRITE HEADING-LINE AFTER ADVANCING PAGE.
001190
001200 020-WRITE-RESPONSES.
001210 MOVE SPACES TO DETAIL-LINE.
001220 MOVE WS-QUESTION-SUB TO QUESTION.
001230
001240 MOVE ANSWER (WS-QUESTION-SUB, 1) TO PRINT-YES.
001250 MOVE ANSWER (WS-QUESTION-SUB, 2) TO PRINT-NO.
001260 MOVE ANSWER (WS-QUESTION-SUB, 3) TO PRINT-NOT-SURE.
001270
001280 WRITE DETAIL-LINE AFTER ADVANCING 1 LINE.

```

Performed routine is executed ten times

ANSWER is referenced with two subscripts

Determines appropriate column for two-level table

FIGURE 10.10 Continued

statement, which causes the paragraph 005-INITIALIZE-SURVEY-TOTALS to be executed 30 times, once for each of 30 elements. The routine 010-PROCESS-SURVEYS is performed until there are no more surveys to process. It in turn invokes the routine 015-EVALUATE-RESPONSE 10 times, i.e., once for each question on the survey. (The PERFORM VARYING of lines 970-990 automatically increments the value of WS-QUESTION-SUB from 1 to 10.) The nested IF of lines 1040-1100 determines the appropriate column in which to enter the particular response, i.e., yes, no, or not sure for columns 1, 2, and 3, respectively. When the end of file is reached, and the perform of lines 800 and 810 is terminated, the computed responses are printed by the routine 020-WRITE-RESPONSES.

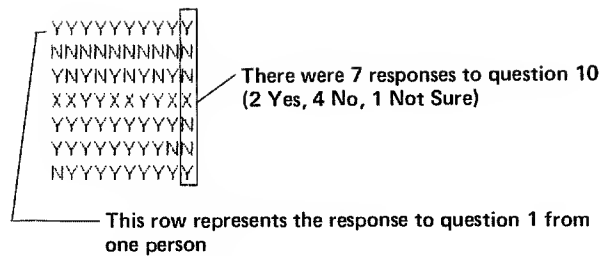


FIGURE 10.11 Input to survey program

Figure 10.11 contains test data and Figure 10.12 corresponding output. There were 7 people who responded to the survey. Each record of Figure 10.11 contains 10 responses; i.e., the answers to questions 1-10 from the same individual. Figure 10.12 contains actual output produced by the COBOL program.

QUESTION	YES	NO	NOT SURE
1	4	2	1
2	4	2	1
3	6	1	
4	5	2	
5	5	1	1
6	4	2	1
7	6	1	
8	5	2	
9	4	2	1
10	2	4	1

A 10 X 3 table (10 rows, 3 columns)

FIGURE 10.12 Output from survey program

THREE-LEVEL TABLES

COBOL permits tables of one, two, or three dimensions. We have covered the first two cases in some detail, and have illustrated both with complete programs. We now provide a brief look at three-level tables.

Three-dimension tables require three subscripts to specify a particular entry. Consider a university with three colleges, five schools (e.g., engineering, business, etc.) within each college, and four years within each school. We define a three-dimension table of enrollments as follows:

```

01 ENROLLMENTS.
   05 COLLEGE OCCURS 3 TIMES.
      10 SCHOOL OCCURS 5 TIMES.
         15 YEAR OCCURS 4 TIMES          PIC 9(4).

```

There are 60 (3×5×4) elements in the table. Note that YEAR is the only elementary item, and hence it is the only entry with a picture clause. The COBOL compiler allocates a total of 240 positions (60 elements × 4 positions per element), as indicated in Figure 10.13.

As can be inferred from Figure 10.13, table positions 1 to 80 refer to the first college, positions 81 to 160 to the second college, and positions 161 to 240 to the third college. Positions 1 to 16 refer to the first school in the first college, positions 81 to 96 refer to the first school in the second college, and positions 161 to 176 refer to the first school in the third college.

PERFORM VARYING

The VARYING option of the PERFORM verb is extremely convenient for manipulating subscripts and/or indexes. It is extended to three dimensions.

PERFORM procedure-name-1 [THRU procedure-name-2]

VARYING { identifier-1
 index-1 } FROM { identifier-2
 literal-2
 index-2 } BY { identifier-3
 literal-3 } UNTIL condition-1

[AFTER { identifier-4
 index-4 } FROM { identifier-5
 literal-5
 index-5 } BY { identifier-6
 literal-6 } UNTIL condition-2]

[AFTER { identifier-7
 index-7 } FROM { identifier-8
 literal-8
 index-8 } BY { identifier-9
 literal-9 } UNTIL condition-3]

As an illustration, consider the following PERFORM statement:

```
PERFORM 010-READ-ENROLLMENT-RECORDS
  VARYING COLLEGE-SUB
    FROM 1 BY 1 UNTIL COLLEGE-SUB > 3
  AFTER SCHOOL-SUB
    FROM 1 BY 1 UNTIL SCHOOL-SUB > 5
  AFTER YEAR-SUB
    FROM 1 BY 1 UNTIL YEAR-SUB > 4.
```

The procedure 010-READ-ENROLLMENT-RECORDS will be performed a total of 60 times, with the *bottom* subscript, YEAR-SUB, varied first. Initially, COLLEGE-SUB, SCHOOL-SUB, and YEAR-SUB are all set to 1, and the first perform is done. Then YEAR-SUB is incremented by 1 and becomes 2 (COLLEGE-SUB and SCHOOL-SUB remain at 1), and a second perform is done. YEAR-SUB is incremented to 3 and then to 4, resulting in two additional performs. YEAR-SUB temporarily becomes 5, but no perform is realized since YEAR-SUB > 4. SCHOOL-SUB is then incremented to 2, YEAR-SUB drops to 1, and we go merrily on our way.

SUMMARY This chapter was devoted entirely to table processing. It began with a review of the OCCURS clause, then expanded the material to indexing and variable-length tables. Three distinct methods for table initialization were presented: hard coding it in a program, use of the COPY statement, and dynamically reading a table. Table lookups were presented, with and without indexes, and with and without PERFORM VARYING. All material was neatly summarized in a complete program.

Coverage was extended to two-level tables. We saw the need for two OCCURS clauses and PERFORM VARYING in two dimensions. This material was highlighted in a second complete program.

Finally, coverage was extended to three-level tables, with emphasis on three OCCURS clauses and PERFORM VARYING in three dimensions.

TRUE/FALSE

1. Initializing a table via a COPY clause is superior to "hard coding."
2. A two-level table requires two OCCURS clauses in its definition.
3. Initializing a table by reading values from a file is superior to using a COPY statement.
4. A single PERFORM statement can manipulate two subscripts.
5. The same COBOL entry can contain both a VALUE and an OCCURS clause.
6. The REDEFINES clause *must* be used when defining a table.
7. The COBOL entries TABLE (4, 1) and TABLE (1, 4) are equivalent.
8. An OCCURS clause allocates space to a table *and* assigns values to it.
9. Direct access to table entries is faster than a sequential table lookup.
10. A two-level table may be referenced at different hierarchical levels.

EXERCISES

1. (a) Write out the 12 pairs of values that will be assumed by SUB-1 and SUB-2 as a result of the statement

```

PERFORM 10-READ-CARDS
  VARYING SUB-1 FROM 1 BY 1
    UNTIL SUB-1 > 4
  AFTER SUB-2 FROM 1 BY 1
    UNTIL SUB-2 > 3.

```

- (b) What would happen if the greater than signs were replaced by equal signs? By less than signs?
2. How many storage positions are allocated for each of the following table definitions? Show an appropriate schematic indicating storage assignment for each table.

```

(a) 01 ENROLLMENTS.
      05 COLLEGE OCCURS 4 TIMES.
      10 SCHOOL OCCURS 5 TIMES.
      15 YEAR OCCURS 4 TIMES      PIC 9(4).

```

```

(b) 01 ENROLLMENTS.
      05 COLLEGE OCCURS 4 TIMES.
      10 SCHOOL OCCURS 5 TIMES      PIC 9(4).
      10 YEAR OCCURS 4 TIMES        PIC 9(4).

```

3. Given the entries:

```

01 NUMBER-OF-EMPLOYEES-TABLE.
      05 REGION OCCURS 6 TIMES.
      10 CITY OCCURS 4 TIMES      PIC 9(3).

```

Indicate whether the following references are valid:

- (a) REGION (1)
- (b) CITY (3, 2)
- (c) REGION (3, 2)
- (d) REGION (7)
- (e) CITY (2)
- (f) NUMBER-OF-EMPLOYEES-TABLE
- (g) CITY (6, 4)
- (h) CITY (4, 6)

Note that some entries may be syntactically valid (i.e., they will not cause compilation errors), but logically invalid. Distinguish between the two kinds of errors.

4. Write out the 24 pairs of values that will be assumed by SUB-1, SUB-2, and SUB-3 as a result of the statement:

```

PERFORM 10-READ-CARDS
  VARYING SUB-3 FROM 1 BY 1
    UNTIL SUB-3 > 3
  AFTER SUB-2 FROM 1 BY 1
    UNTIL SUB-2 > 2
  AFTER SUB-1 FROM 1 BY 1
    UNTIL SUB-1 > 4.

```

5. *Debugging*—Figure 10.14 contains invalid output produced by the Table Processing program of Figure 10.15. (Figure 10.6, and the associated discussion, presented processing specifications, test data, and intended output.) Find and correct all errors in Figure 10.15 which represents a modified version of the program of Figure 10.5.

Location lookup fails

PERSONNEL PROFILE

NAME: JONES

LOCATION: UNKNOWN

EDUCATION: 4YR DEGREE

SOC SEC NUM: 222-22-2222

TITLE: UNKNOWN

PERFORMANCE: EXCELLENT

SALARY DATA

SALARY	DATE	% INCREASE	MBI
\$23,000	3/81	15.00	5
\$20,000	9/80		

Title lookup fails

Months between increase is wrong

PERSONNEL PROFILE

NAME: SMITH

LOCATION: UNKNOWN

EDUCATION: SOME GRAD

SOC SEC NUM: 111-11-1111

TITLE: UNKNOWN

PERFORMANCE: UNKNOWN

Performance should be average

SALARY DATA

SALARY	DATE	% INCREASE	MBI
\$18,000	7/81		

FIGURE 10.14 Invalid output of table program

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID. ETABLES.
000120 AUTHOR. R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER. TRS-80.
000170 OBJECT-COMPUTER. TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210 SELECT TITLE-FILE
000220 ASSIGN TO INPUT "TITLES/DAT".
000230 SELECT EMPLOYEE-FILE
000240 ASSIGN TO INPUT "TABLES/DAT".
000250 SELECT PRINT-FILE
000260 ASSIGN TO PRINT "ETABLES/TXT".
000270
000280 DATA DIVISION.
000290 FILE SECTION.
000300 FD TITLE-FILE
000310 LABEL RECORDS ARE STANDARD
000320 RECORD CONTAINS 20 CHARACTERS
000330 DATA RECORD IS SALES-RECORD.
000340 01 TITLE-RECORD.
000350 05 IN-TITLE-CODE PIC 9(3).
000360 05 IN-TITLE-VALUE PIC X(17).
000370
000380 FD EMPLOYEE-FILE
000390 LABEL RECORDS ARE STANDARD
000400 RECORD CONTAINS 50 CHARACTERS
000410 DATA RECORD IS EMPLOYEE-RECORD.
000420 01 EMPLOYEE-RECORD.
000430 05 EMP-SOC-SEC-NUMBER PIC 9(9).
000440 05 EMP-LAST-NAME PIC X(15).
000450 05 EMP-LOC-CODE PIC X(3).
000460 05 EMP-TITLE-CODE PIC 999.
000470 05 EMP-EDUCATION-CODE PIC 9.
000480 05 EMP-PERFORMANCE-CODE PIC X.
000490 05 EMP-SALARY-DATA OCCURS 2 TIMES.
000500 10 EMP-SALARY PIC 9(5).
000510 10 EMP-SALARY-MONTH PIC 99.
000520 10 EMP-SALARY-YEAR PIC 99.
000530
000540 FD PRINT-FILE
000550 LABEL RECORDS ARE STANDARD
000560 RECORD CONTAINS 132 CHARACTERS
000570 DATA RECORD IS PRINT-LINE.
000580 01 PRINT-LINE PIC X(132).
000590
000600 WORKING-STORAGE SECTION.
000610 01 NUMBER-OF-TITLES PIC 9(3) VALUE ZEROS.
000620
000630 01 PROGRAM-SWITCHES.
000640 05 EMPLOYEE-FILE-SWITCH PIC X(3) VALUE SPACES.
000650 05 TITLE-FILE-SWITCH PIC X(3) VALUE SPACES.
000660 05 FOUND-LOCATION-SWITCH PIC X(3) VALUE SPACES.
000670 05 FOUND-TITLE-SWITCH PIC X(3) VALUE SPACES.
000680
000690 01 PROGRAM-SUBSCRIPTS.
000700 05 LOCATION-SUB PIC 99.
000710 05 TITLE-SUB PIC 99.
000720
000730 01 SALARY-CALCULATIONS.
000740 05 PERCENT-SALARY-INCREASE PIC 99V9.
000750 05 MONTHS-BETWEEN-INCREASE PIC 99.
000760
000770 01 LOCATION-VALUES.
000780 05 FILLER PIC X(18) VALUE "ATLANTA"
000790 05 FILLER PIC X(18) VALUE "BOSTON"
000800 05 FILLER PIC X(18) VALUE "CHICAGO"
000810 05 FILLER PIC X(18) VALUE "DETROIT"

```

FIGURE 10.15 Incorrect table program

```

000820      05 FILLER                      PIC X(18) VALUE "LOULOUISVILLE  ".
000830      05 FILLER                      PIC X(18) VALUE "MINMINNEAPOLIS  ".
000840      05 FILLER                      PIC X(18) VALUE "NEWNEWARK      ".
000850      05 FILLER                      PIC X(18) VALUE "NY NEW YORK      ".
000860      05 FILLER                      PIC X(18) VALUE "SA SAN ANTONIO   ".
000870      05 FILLER                      PIC X(18) VALUE "SF SAN FRANCISCO".
000880
000890 01 LOCATION-TABLE.
000900      05 LOCATION-CODE-AND-VALUE OCCURS 10 TIMES.
000910          10 LOCATION-CODE          PIC X(3).
000920          10 EXPANDED-LOCATION PIC X(15).
000930
000940 COPY "EDUCATE/CBL".
000001 01 EDUCATION-TABLE.
000002      05 EDUCATION-VALUES.
000003          10 FILLER                      PIC X(10) VALUE "SOME H.S.  ".
000004          10 FILLER                      PIC X(10) VALUE "HS DIPLOMA".
000005          10 FILLER                      PIC X(10) VALUE "2YR DEGREE".
000006          10 FILLER                      PIC X(10) VALUE "4YR DEGREE".
000007          10 FILLER                      PIC X(10) VALUE "SOME GRAD  ".
000008          10 FILLER                      PIC X(10) VALUE "MASTERS   ".
000009          10 FILLER                      PIC X(10) VALUE "PH. D.    ".
000010          10 FILLER                      PIC X(10) VALUE "OTHER    ".
000011
000012      05 EDU-NAME REDEFINES EDUCATION-VALUES
000013          OCCURS 8 TIMES PIC X(10).
000014
000950
000960 01 TITLE-TABLE.
000970      05 TITLES OCCURS 1 TO 100 TIMES
000980          DEPENDING ON NUMBER-OF-TITLES
000990          INDEXED BY TITLE-INDEX.
001000          10 TITLE-CODE          PIC 9(3).
001010          10 TITLE-VALUE         PIC X(17).
001020
001030 01 PRINT-LINE-ONE.
001040      05 FILLER                      PIC X(22) VALUE SPACES.
001050      05 FILLER                      PIC X(17)
001060          VALUE "PERSONNEL PROFILE".
001070      05 FILLER                      PIC X(93) VALUE SPACES.
001080
001090 01 PRINT-LINE-TWO.
001100      05 FILLER                      PIC X(6)  VALUE "NAME:  ".
001110      05 PRT-LAST-NAME              PIC X(15).
001120      05 FILLER                      PIC X(25) VALUE SPACES.
001130      05 FILLER                      PIC X(13) VALUE "SOC SEC NUM: ".
001140      05 PRT-SOC-SEC-NUMBER         PIC 999B99B9999.
001150      05 FILLER                      PIC X(62) VALUE SPACES.
001160
001170 01 PRINT-LINE-THREE.
001180      05 FILLER                      PIC X(10) VALUE "LOCATION: ".
001190      05 PRT-LOCATION                PIC X(15).
001200      05 FILLER                      PIC X(21) VALUE SPACES.
001210      05 FILLER                      PIC X(7)  VALUE "TITLE: ".
001220      05 PRT-TITLE                  PIC X(17).
001230      05 FILLER                      PIC X(62) VALUE SPACES.
001240
001250 01 PRINT-LINE-FOUR.
001260      05 FILLER                      PIC X(11) VALUE "EDUCATION: ".
001270      05 PRT-EDUCATION              PIC X(10).
001280      05 FILLER                      PIC X(25) VALUE SPACES.
001290      05 FILLER                      PIC X(13) VALUE "PERFORMANCE: ".
001300      05 PRT-PERFORMANCE            PIC X(11).
001310      05 FILLER                      PIC X(62) VALUE SPACES.
001320
001330 01 PRINT-LINE-FIVE.
001340      05 FILLER                      PIC X(25) VALUE SPACES.
001350      05 FILLER                      PIC X(11) VALUE "SALARY DATA".
001360      05 FILLER                      PIC X(96) VALUE SPACES.
001370
001380 01 PRINT-LINE-SIX.
001390      05 FILLER                      PIC X(5)  VALUE SPACES.

```

FIGURE 10.15 *Continued*

```

001400      05  FILLER                      PIC X(6)  VALUE "SALARY".
001410      05  FILLER                      PIC X(10) VALUE SPACES.
001420      05  FILLER                      PIC X(4)  VALUE "DATE".
001430      05  FILLER                      PIC X(10) VALUE SPACES.
001440      05  FILLER                      PIC X(10) VALUE "% INCREASE".
001450      05  FILLER                      PIC X(10) VALUE SPACES.
001460      05  FILLER                      PIC X(3)  VALUE "MBI".
001470      05  FILLER                      PIC X(74) VALUE SPACES.
001480
001490  01  PRINT-SALARY-LINE.
001500      05  FILLER                      PIC X(4)  VALUE SPACES.
001510      05  PRT-SALARY                   PIC $99,999.
001520      05  FILLER                      PIC X(9)  VALUE SPACES.
001530      05  PRT-SALARY-MONTH            PIC Z9.
001540      05  PRT-SLASH                   PIC X      VALUE "/".
001550      05  PRT-SALARY-YEAR             PIC Z9.
001560      05  FILLER                      PIC X(12) VALUE SPACES.
001570      05  PRT-SALARY-INCREASE         PIC Z9.99.
001580      05  FILLER                      PIC X(13) VALUE SPACES.
001590      05  PRT-SALARY-MBI             PIC ZZ.
001600      05  FILLER                      PIC X(75) VALUE SPACES.
001610
001620  PROCEDURE DIVISION.
001630  010-PREPARE-REPORT.
001640      PERFORM 020-INITIALIZE-TITLE-TABLE.
001650
001660      OPEN INPUT EMPLOYEE-FILE
001670          OUTPUT PRINT-FILE.
001680      READ EMPLOYEE-FILE
001690          AT END MOVE "YES" TO EMPLOYEE-FILE-SWITCH.
001700      PERFORM 040-WRITE-PERSONNEL-PROFILES
001710          UNTIL EMPLOYEE-FILE-SWITCH = "YES".
001720      CLOSE EMPLOYEE-FILE
001730          PRINT-FILE.
001740      STOP RUN.
001750
001760  020-INITIALIZE-TITLE-TABLE.
001770      OPEN INPUT TITLE-FILE.
001780      READ TITLE-FILE
001790          AT END MOVE "YES" TO TITLE-FILE-SWITCH.
001800      PERFORM 030-READ-TITLE-FILE
001810          VARYING TITLE-SUB FROM 1 BY 1
001820          UNTIL TITLE-FILE-SWITCH = "YES".
001830      CLOSE TITLE-FILE.
001840
001850  030-READ-TITLE-FILE.
001860      IF TITLE-SUB > 100
001870          MOVE "YES" TO TITLE-FILE-SWITCH
001880          DISPLAY "ERROR - TITLE TABLE EXCEEDED"
001890      ELSE
001910          MOVE IN-TITLE-CODE TO TITLE-CODE (TITLE-SUB)
001920          MOVE IN-TITLE-VALUE TO TITLE-VALUE (TITLE-SUB).
001930      READ TITLE-FILE
001940          AT END MOVE "YES" TO TITLE-FILE-SWITCH.
001950
001960  040-WRITE-PERSONNEL-PROFILES.
001970      WRITE PRINT-LINE FROM PRINT-LINE-ONE
001980          AFTER ADVANCING PAGE.
001990
002000      MOVE EMP-LAST-NAME TO PRT-LAST-NAME.
002010      MOVE EMP-SOC-SEC-NUMBER TO PRT-SOC-SEC-NUMBER.
002020      INSPECT PRT-SOC-SEC-NUMBER
002030          REPLACING ALL " " BY "-".
002040      WRITE PRINT-LINE FROM PRINT-LINE-TWO
002050          AFTER ADVANCING 2 LINES.
002060
002070      MOVE "NO" TO FOUND-LOCATION-SWITCH.
002080      PERFORM 050-SEARCH-LOCATION-TABLE
002090          VARYING LOCATION-SUB FROM 1 BY 1
002100          UNTIL FOUND-LOCATION-SWITCH = "YES".
002110
002120      MOVE "NO" TO FOUND-TITLE-SWITCH.

```

FIGURE 10.15 *Continued*


```

002130      SET TITLE-INDEX TO 1.
002140      PERFORM 060-SEARCH-TITLE-TABLE
002150          UNTIL FOUND-TITLE-SWITCH = "YES".
002160
002170      WRITE PRINT-LINE FROM PRINT-LINE-THREE
002180          AFTER ADVANCING 2 LINES.
002190
002200      PERFORM 070-EXPAND-EDUCATION-CODE.
002210      PERFORM 080-EXPAND-PERFORMANCE-CODE.
002220      WRITE PRINT-LINE FROM PRINT-LINE-FOUR
002230          AFTER ADVANCING 2 LINES.
002240
002250      WRITE PRINT-LINE FROM PRINT-LINE-FIVE
002260          AFTER ADVANCING 5 LINES.
002270
002280      WRITE PRINT-LINE FROM PRINT-LINE-SIX
002290          AFTER ADVANCING 2 LINES.
002300
002310      MOVE SPACES TO PRINT-SALARY-LINE.
002320      MOVE "/" TO PRT-SLASH.
002330      MOVE EMP-SALARY (1) TO PRT-SALARY.
002340      MOVE EMP-SALARY-MONTH (1) TO PRT-SALARY-MONTH.
002350      MOVE EMP-SALARY-YEAR (1) TO PRT-SALARY-YEAR.
002360
002370      IF EMP-SALARY (2) > 0
002380          PERFORM 090-EVALUATE-SALARY-INCREASES
002390          MOVE PERCENT-SALARY-INCREASE TO PRT-SALARY-INCREASE
002400          MOVE MONTHS-BETWEEN-INCREASE TO PRT-SALARY-MBI
002410          WRITE PRINT-LINE FROM PRINT-SALARY-LINE
002420              AFTER ADVANCING 2 LINES
002430          MOVE SPACES TO PRINT-SALARY-LINE
002440          MOVE "/" TO PRT-SLASH
002450          MOVE EMP-SALARY (2) TO PRT-SALARY
002460          MOVE EMP-SALARY-MONTH (2) TO PRT-SALARY-MONTH
002470          MOVE EMP-SALARY-YEAR (2) TO PRT-SALARY-YEAR.
002480      WRITE PRINT-LINE FROM PRINT-SALARY-LINE
002490          AFTER ADVANCING 1 LINE.
002500      READ EMPLOYEE-FILE
002510          AT END MOVE "YES" TO EMPLOYEE-FILE-SWITCH.
002520
002530 050-SEARCH-LOCATION-TABLE.
002540      IF LOCATION-SUB > 10
002550          MOVE "UNKNOWN" TO PRT-LOCATION
002560          MOVE "YES" TO FOUND-LOCATION-SWITCH
002570      ELSE
002580          IF LOCATION-CODE (LOCATION-SUB) = EMP-LOC-CODE
002590              MOVE EXPANDED-LOCATION (LOCATION-SUB) TO PRT-LOCATION
002600              MOVE "YES" TO FOUND-LOCATION-SWITCH.
002610
002620 060-SEARCH-TITLE-TABLE.
002630      IF TITLE-INDEX > NUMBER-OF-TITLES
002640          MOVE "UNKNOWN" TO PRT-TITLE
002650          MOVE "YES" TO FOUND-TITLE-SWITCH
002660      ELSE
002670          IF TITLE-CODE (TITLE-INDEX) = EMP-TITLE-CODE
002680              MOVE TITLE-VALUE (TITLE-INDEX) TO PRT-TITLE
002690              MOVE "YES" TO FOUND-TITLE-SWITCH
002700      ELSE
002710          SET TITLE-INDEX UP BY 1.
002720
002730 070-EXPAND-EDUCATION-CODE.
002740      IF EMP-EDUCATION-CODE < 1 OR EMP-EDUCATION-CODE > 8
002750          MOVE "UNKNOWN" TO PRT-EDUCATION
002760      ELSE
002770          MOVE EDU-NAME (EMP-EDUCATION-CODE) TO PRT-EDUCATION.
002780
002790 080-EXPAND-PERFORMANCE-CODE.
002800      IF EMP-PERFORMANCE-CODE = "E"
002810          MOVE "EXCELLENT" TO PRT-PERFORMANCE
002820      ELSE

```

FIGURE 10.15 *Continued*

```

002830      IF EMP-PERFORMANCE-CODE = "A"
002840          MOVE "AVERAGE" TO PRT-PERFORMANCE
002860      IF EMP-PERFORMANCE-CODE = "P"
002870          MOVE "POOR" TO PRT-PERFORMANCE
002880      ELSE
002890          MOVE "UNKNOWN" TO PRT-PERFORMANCE.
002900
002910 090-EVALUATE-SALARY-INCREASES.
002920      COMPUTE PERCENT-SALARY-INCREASE
002930          = 100 * (EMP-SALARY (1) - EMP-SALARY (2)) / EMP-SALARY (2).
002940
002950      COMPUTE MONTHS-BETWEEN-INCREASE
002960          = (EMP-SALARY-YEAR (1) - EMP-SALARY-YEAR (2))
002970          + (EMP-SALARY-MONTH (1) - EMP-SALARY-MONTH (2)).

```

FIGURE 10.15 *Continued*

PROJECTS

1. A large photo album company sells photographs to school children in two counties. They collect the following information for each student: name, grade, a preassigned number for the school, and an indication of up to three types of picture package(s) the parents wish to buy. There are seven types of packages, numbers 1 to 7, but the parent buys only *one* of each type desired. The input record contains the following data:

Columns	Field	Picture
1-25	NAME	X(25)
26-29	SCHOOL	X(4)
31-32	GRADE	XX
34	PACKAGE-1	9
36	PACKAGE-2	9
38	PACKAGE-3	9

The values appearing in columns 34, 36, and 38 indicate the type (*not quantity*) of respective picture package(s); e.g., a 4 in column 34, a 6 in column 36, and a blank in column 38 indicate that the parent will buy 1 package of type 4 and 1 package of type 6. Your job is to write a program to print the total amount due from individuals who have purchased pictures.

Do not print the names of children ordering no packages. For those students ordering packages, i.e., those with entries in columns 34, 36, or 38, check for a valid number from 1 to 7 (use an 88-level entry with a VALUES ARE clause). If the number is invalid, display a message and the person's name, school, and grade.

If the value in column 34, 36, or 38 is between 1 and 7, use it as a subscript and move the appropriate cost from the following table. Code this table into your program.

Package Number	Cost
1	\$7.50
2	\$4.00
3	\$3.75
4	\$2.50
5	\$2.50
6	\$8.00
7	\$10.50

Chapter 10

Print the total due for each child as it occurs as well as a grand total on a total line.

The following test data are provided:

WATSON, AMY	0023 02 1 7
HOLMES, ANDREW	0023 02 1
MITCHUM, MARY	0023 03 2
HEDIN, SAM	0023 04 3 4 5
FLETCHER, RAYMOND	0023 04
FOSTER, ED	0023 05 1 8
HARDING, ZACHARY	0023 07 2
JACOBSON, SUE	0023 07 6 1

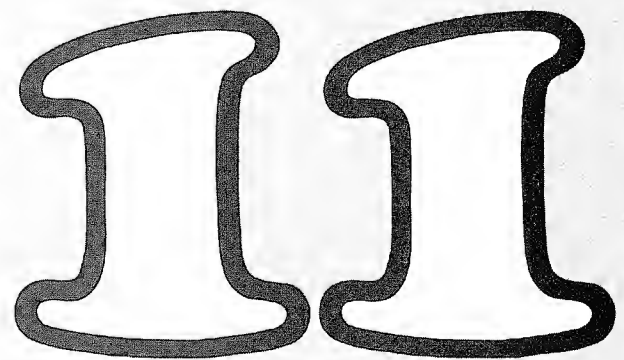
2. Acme Widgets, Inc., has branch offices in New York, Los Angeles, and Miami (locations 1, 2, and 3, respectively). Each location has five departments: 10, 11, 12, 13, and 14. A file has been established which contains a record for each employee in Acme containing the following information:

<i>Field</i>	<i>Columns</i>	<i>Picture</i>
NAME	1-20	X(20)
LOCATION	21	X
DEPARTMENT	22-23	99

Write a COBOL program to read the file, compute, and print:

- (a) The total number of employees in each location.
- (b) The total number of employees in each department throughout the company (five totals in all).
- (c) The total number of employees in each department in each location (15 totals in all).

Use any suitable format to print the totals; make up your own test data.



**SEQUENTIAL
FILE
MAINTENANCE**

OVERVIEW Virtually all of the programs shown so far have produced a report of one type or another. The results of this kind of data processing activity are very visible to the small business, as they contain information relating directly to the business; i.e., sales activity, accounts receivable, payroll, etc. Realize, however, that the files on which these reports are based must be *maintained* to reflect the changing nature of the physical environment. File maintenance, therefore, is the subject of this chapter and the next.

Files can be maintained either *sequentially* or *nonsequentially*. Sequential maintenance means that *every* record is accessed in a given run, whereas nonsequential maintenance accesses *only* those records which are actually changed. Sequential processing is discussed in this chapter and nonsequential processing is deferred to Chapter 12. (A more advanced discussion of file maintenance may be found in R. Grauer, *Structured Methods Through COBOL*, Prentice-Hall, Inc. 1983.)

The logic required for file maintenance, be it sequential or nonsequential, is rather complex. Accordingly, considerable space is devoted to the design and testing of logically involved programs. Our coverage includes pseudocode, hierarchy charts, and top down testing. We begin, however, with a conceptual discussion of file maintenance.

CONCEPTS OF FILE MAINTENANCE

Every application must provide for *additions, deletions, or changes* to existing records. In a payroll system, for example, new employees can be added, while existing employees can be terminated or receive salary increases. A sequential file update is shown schematically in Figure 11.1. Input to the sequential update consists of two files; an old master and a transaction file. The latter contains different kinds of transactions to accommodate various changes to the physical system. Output from the update is a new master file and a series of reports reflecting the update.

Sequential maintenance is usually done periodically. Let us assume a payroll system is updated monthly and begins with a current master file on January 1. Transactions are collected (batched) during the month of January. Then, on February 1, we take the master file as of January 1 (now the old master), the transactions accrued during January, and produce a new master as of February 1. The process continues from month to month. Transactions are collected during February. On March 1, we take the file created February 1 as the old master, run it against the February transactions, and produce a new master as of March 1. Figure 11.2 illustrates this discussion.

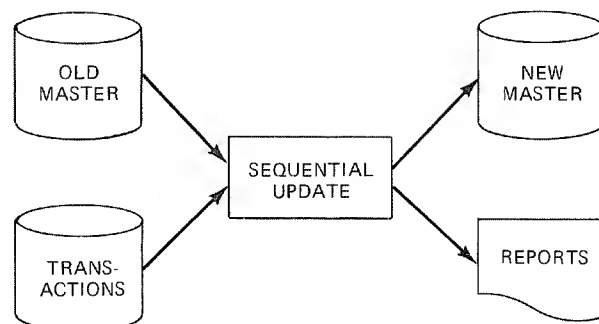


FIGURE 11.1 Sequential file update

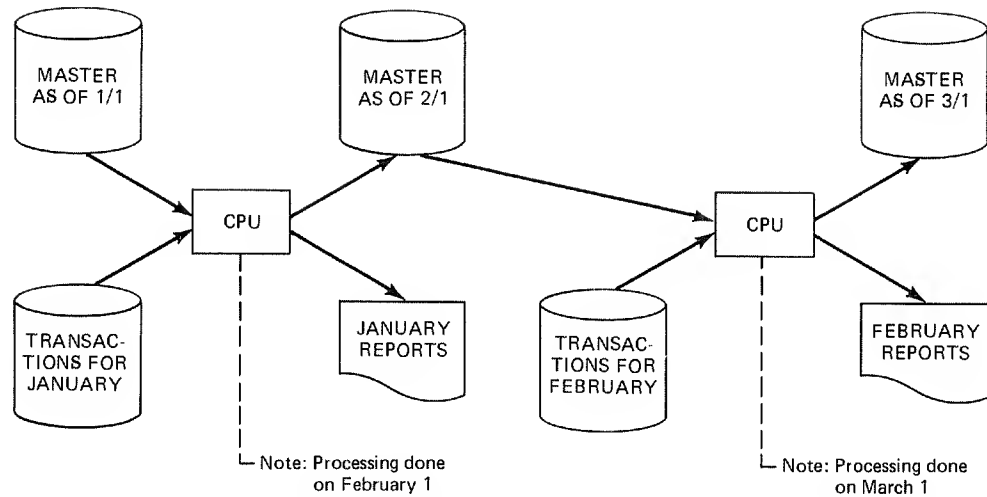


FIGURE 11.2 Two-period sequential update

In Figure 11.2, the master files for January, February, and March are physically different files. Further, since the January master gave rise to the February master, which in turn spawned the March master, the files are known as grandfather, father, and son, respectively.

MURPHY'S LAW

Murphy's law states that "if something can go wrong, it will—and at the worst possible time." (A corollary, by an unknown author, is that Murphy was an optimist.) The statement is particularly true for data processing, and it is absolutely essential that one provide as much *backup* as feasible. What if the diskette containing the February master were inadvertently destroyed on February 27, just prior to the next update? If one were prudent enough to keep both the January 1 master and transactions for January, it would be a simple matter to recreate the February 1 master file. A generally accepted practice is to keep at least the last three historical levels, i.e., the grandfather, father, and son, for backup purposes.

How could information on a diskette be destroyed in the first place? First and foremost, by human error. It takes only a second to accidentally enter the system command, "KILL OLDMAST", and the file is irretrievably gone. A second way is through some type of hardware problem; e.g., a defective or worn diskette, a malfunctioning disk drive, a temporary loss of power, etc. Even though the programmer is not directly responsible, and indeed did not cause such hardware problems, he or she still suffers the consequences. *Anticipation* is the key word. Assume things will go wrong and keep *multiple* copies of all data files (as well as multiple copies of all programs) on *different* diskettes, in *different* locations.

The price of the extra diskettes and/or the time required to run the TRS-80 BACKUP utility is a small one compared to losing a file and having no backup.

REQUIREMENTS OF THE MAINTENANCE PROGRAM

Given an overall view of the maintenance process, and the need for backup procedures, we now focus on specifics of the COBOL program. As can be seen from Figure 11.1, the maintenance program accepts an old master and a transaction file as input, and produces a new master file and various reports as output.

The transaction file must provide the ability to add new records to the master file, as well as delete or alter existing records. The exact nature of the transaction is indicated by a transaction code. The record layout of the old master and transaction files is seen in Figure 11.3. (The record layout of the new master is identical to that of the old master.)

Four types of transactions are possible as indicated by the 88-level entries in Figure 11.3b. The nature of each transaction type is explained as follows:

- A — Addition: Indicates a new record is to be added to the master file. This transaction type requires name, social security number, and *all* other fields.
- C — Correction: Indicates that one or more fields in an existing master record is to be corrected. One enters the social security number and only those fields that require correction; if a given field is correct in the master file, it should not appear on the transaction record.
- D — Deletion: Indicates that a record currently in the master file is to be deleted from the new master. (Deleted records, however, are to be written to a separate file for possible recall at a later date.) Enter name and social security number only.
- U — Salary Update: Indicates that an individual received a salary increase. The *present* salary on the old master becomes the *previous* salary on the new master, causing *transaction* salary to become the *present* salary on the new master. (Note well that two levels of salary data are present in the old master record as per the OCCURS clause of Figure 11.3a.)

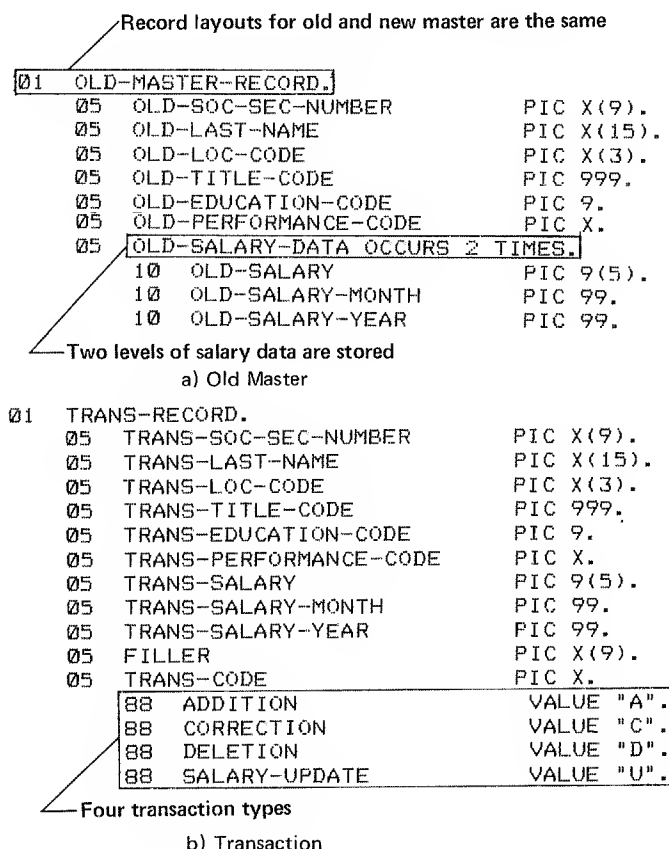


FIGURE 11.3 Old master and transaction records

Murphy's law should also be considered in the maintenance procedure. Although the previous instructions for completing transactions are complete and unambiguous, data entry personnel may inadvertently introduce errors. Hence, the transactions should first be verified in a "stand alone" edit program to assure that they have been coded correctly. The edit program should check that each transaction has a valid transaction code; either A, C, D, or U. It should verify that the transactions are in sequential order by social security number. It should ascertain that all fields have been entered for an addition, and so on. In practice, the edit program may have to be run several times until it accepts all incoming transactions as valid. Only when this has been accomplished should the maintenance program be attempted. (The chapter summary lists typical programming checks which may be included in the "stand alone" edit program.)

Even when the transaction file has been accepted by the edit program as correct, it may still contain invalid transactions. That is because the edit program considers the transaction file as a separate entity and cannot check on its relationship to the master file. What happens, for example, if the transaction social security number for an attempted correction to an existing master record is miscopied? Accordingly, the file maintenance program itself must contain *additional* error checks which could not be included in the stand alone edit program. These include:

1. "No Matches" resulting from miscopied social security numbers for transaction types C, D, or U,
2. "Duplicate Adds" which result when the social security number of an attempted addition is already present in the old master file, and
3. "Duplicate Salary Updates", which occur when the transaction salary on a U type transaction already exists in the old master.

The logic to accomplish the required processing is relatively complex. Accordingly, we develop the corresponding pseudocode to aid in the presentation.

PSEUDOCODE Figure 11.4 contains pseudocode for the sequential update. Recall that pseudocode is an alternative to the more traditional flowchart as a means of expressing a program's decision making logic. Realize also that in sequential file maintenance, input is taken from *two* places, the old master and transaction files, and that the action taken depends on the *relationship* between the most recently read records.

If the social security number on the old master is less than the social security number on the transaction file, it means there is no activity for the current old master record. Hence, it should be merely copied to the new master file. If the social security numbers are equal, then additional processing is required to determine the type of transaction. An addition in this instance would signal an error, i.e., a "duplicate add", whereas a C, D, or U requires additional processing.

Finally, if the old master social security number is greater than the transaction social security number, it means the current transaction is not on the existing master file. This in turn implies either a new record is to be added, or the social security number of an existing record was miscopied. A transaction code of A in this instance should be processed as a valid addition, whereas a code of C, D, or U implies a "no match."

The author believes that the pseudocode of Figure 11.4 is a succinct,


```

Open files
Initial reads for old-master and transaction files
PERFORM until no more data on both files
  IF old-master < transaction
    Copy old-master record
    Read old-master only
  ELSE IF old-master = transaction
    IF addition — write error "Duplicate Record"
    ELSE IF correction — change fields
    ELSE IF deletion — delete record
    ELSE IF update — do salary update
    ENDIF
    Read old-master and transaction files
  ELSE IF old-master > transaction
    IF addition — add record
    ELSE — write error "No match"
    ENDIF
    Read transaction file only
  ENDIF
ENDPERFORM
Close files
Stop run

```

FIGURE 11.4 Pseudocode for sequential update

yet eloquent way to communicate the previous discussion. Consequently, pseudocode is useful as both a design aid and documentation tool. It is a design aid because its availability facilitates coding of the subsequent program. It is also a useful piece of program documentation because it concisely expresses a program's logic.

HIERARCHY CHARTS

The concept of a hierarchy chart was first introduced in Chapter 1. A hierarchy chart is analogous to a company's organization chart, except that the organization is a COBOL program. The hierarchy chart shows the *functions* which have to be accomplished by the program. It also shows the relationship of COBOL paragraphs to each other.

Recall that each box in a hierarchy chart corresponds to a paragraph in the COBOL program. Each paragraph can only be called from the paragraph directly above it, and must return control to that paragraph when its job is completed.

The hierarchy chart for the sequential maintenance program is shown in Figure 11.5. Each box in the hierarchy chart has a distinct job to perform, the nature of which is indicated by the module name. The author endeavors to use strong functional names for his paragraphs, consisting of a verb, adjective or two, and an object; e.g., UPDATE-EMPLOYEE-SALARY, CORRECT-OLD-RECORD, READ-TRANSACTION-FILE, and so on. Indeed, if paragraphs can't be named in this fashion, they may not be functional and thought should be given to possible redesign of the program.

A module's position in a hierarchy chart indicates its importance in the program, just as a person's place in an organization chart says something about his status in the company. The most important modules, i.e., those with the more complex logic, are close to the top; the less important modules, i.e., those with less complex logic, tend to be near the bottom.

A hierarchy chart is also useful as both a design aid and documentation tool. It is helpful before the program is written (in the design phase) to indi-

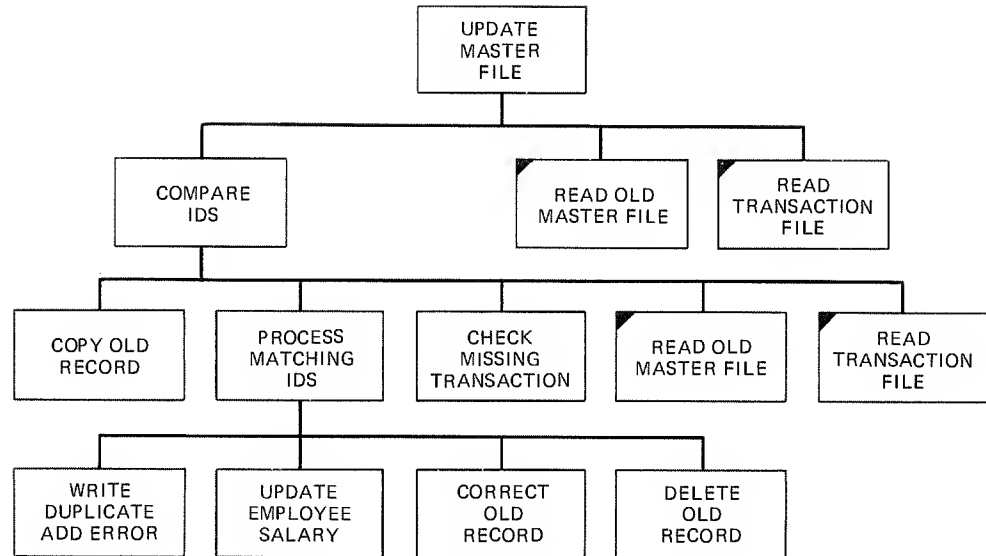


FIGURE 11.5 Hierarchy chart for sequential update

cate the necessary modules or COBOL paragraphs. It is useful after the program has been developed to indicate the relationship of the paragraphs to each other, and the hierarchy of PERFORM statements, within the program. (Boxes shaded in the upper left corner indicate paragraphs which may be called from more than one place in the program).

Consider now the use of the hierarchy chart in program testing, using a methodology known as top down development.

TOP DOWN DEVELOPMENT

A new organization does not rush haphazardly into business by simultaneously hiring all its employees. Rather, the owners of the company will carefully develop an organizational strategy, begin to hire people at the *top* of the organization, and gradually work their way down.

This analogy holds for program development as well. A program should be completely thought out before any coding begins. Moreover, coding should begin at the *top* of the hierarchy chart, with the modules containing the highest levels of logic. It should not begin in the lowest level routines, although many programmers unfortunately start in the wrong place. They become involved, even obsessed, with initially coding unimportant details. They spend, for example, inordinate amounts of time coding Data Division entries for report formatting and the like.

The benefits of the top down approach are best seen in the light of program testing. The traditional approach to testing required that a program be completely coded before any testing could begin. While this may sound logical, it is possible, and *preferable*, to begin testing much earlier and well before coding is finished. This is accomplished by testing the modules in a program's hierarchy chart in the order in which they are developed, that is, from the top down. This is made possible by initially coding lower level modules as *program stubs*, single-sentence routines consisting of a DISPLAY statement to indicate only that the module has been called.

The major advantage of top down testing is that "bugs" are detected earlier and easier than with conventional testing. It ensures that the higher level modules, which typically contain the more complex logic, are tested more frequently than lower level routines. It allows testing and coding to become parallel activities, which provides early feedback to the programmer.

So that you can fully appreciate the significance of top down testing, a partially completed program for sequential maintenance will be developed. This program embodies the logic embedded in the pseudocode of Figure 11.4, and provides for all the modules in the hierarchy chart of Figure 11.5. However, the lowest level routines, UPDATE-EMPLOYEE-SALARY, CORRECT-OLD-RECORD, DELETE-OLD-RECORD, and so on, appear only as program stubs. It will be shown that significant testing can take place *without* providing details of the update, correction, or addition procedures. In other words, one is initially more concerned with the *interaction* of these modules and the higher level routines that call them than with the specifics of updating, correcting, or adding records. Once this interaction has been coded and tested, it becomes almost trivial to complete the details.

A Stubs Program

Figure 11.6 contains a “complete, but incomplete” program for the sequential update. This apparent contradiction in terms is resolved as follows. The program is complete in the sense that every paragraph of the hierarchy chart is accounted for. It is incomplete in that several of the lower level modules consist of a single DISPLAY statement or program stub. In other words, the details for the lower level modules, e.g., lines 1580 through 1680 in Figure 11.6, have been omitted.

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID.    SEQSTUBS.
000120 AUTHOR.        R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER.    TRS-80.
000170 OBJECT-COMPUTER.    TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210     SELECT TRANSACTION-FILE
000220         ASSIGN TO INPUT "TRANS/DAT".
000230     SELECT OLD-MASTER-FILE
000240         ASSIGN TO INPUT "TABLES/DAT".
000250     SELECT NEW-MASTER-FILE
000260         ASSIGN TO OUTPUT "NEWMAS/DAT".
000270     SELECT DELETED-RECORD-FILE
000280         ASSIGN TO OUTPUT "DELETED/DAT".
000290
000300 DATA DIVISION.
000310 FILE SECTION.
000320 FD  TRANSACTION-FILE
000330     LABEL RECORDS ARE STANDARD
000340     RECORD CONTAINS 51 CHARACTERS
000350     DATA RECORD IS TRANS-RECORD.
000360 01  TRANS-RECORD.
000370     05  TRANS-SOC-SEC-NUMBER          PIC X(9).
000380     05  TRANS-LAST-NAME              PIC X(15).
000390     05  TRANS-LOC-CODE               PIC X(3).
000400     05  TRANS-TITLE-CODE            PIC 999.
000410     05  TRANS-EDUCATION-CODE        PIC 9.
000420     05  TRANS-PERFORMANCE-CODE      PIC X.
000430     05  TRANS-SALARY                PIC 9(5).
000440     05  TRANS-SALARY-MONTH          PIC 99.
000450     05  TRANS-SALARY-YEAR           PIC 99.
000460     05  FILLER                     PIC X(9).

```

FIGURE 11.6 Stubs program for sequential update

```

000470      05  TRANS-CODE                                PIC X.
000471      88  ADDITION                                    VALUE "A".
000472      88  CORRECTION                                VALUE "C".
000473      88  DELETION                                    VALUE "D".
000474      88  SALARY-UPDATE                               VALUE "U".
000480
000490 FD  OLD-MASTER-FILE                                Four types of transactions
000500      LABEL RECORDS ARE STANDARD
000510      RECORD CONTAINS 50 CHARACTERS
000520      DATA RECORD IS OLD-MASTER-RECORD.
000530 01  OLD-MASTER-RECORD.
000540      05  OLD-SOC-SEC-NUMBER                          PIC X(9).
000550      05  OLD-LAST-NAME                              PIC X(15).
000560      05  OLD-LOC-CODE                                PIC X(3).
000570      05  OLD-TITLE-CODE                            PIC 999.
000580      05  OLD-EDUCATION-CODE                        PIC 9.
000590      05  OLD-PERFORMANCE-CODE                     PIC X.
000600      05  OLD-SALARY-DATA OCCURS 2 TIMES.
000610          10  OLD-SALARY                            PIC 9(5).
000620          10  OLD-SALARY-MONTH                       PIC 99.
000630          10  OLD-SALARY-YEAR                        PIC 99.
000640
000650 FD  NEW-MASTER-FILE
000660      LABEL RECORDS ARE STANDARD
000670      RECORD CONTAINS 50 CHARACTERS
000680      DATA RECORD IS NEW-MASTER-RECORD.
000690 01  NEW-MASTER-RECORD.
000700      05  NEW-SOC-SEC-NUMBER                          PIC X(9).
000710      05  NEW-LAST-NAME                              PIC X(15).
000720      05  NEW-LOC-CODE                                PIC X(3).
000730      05  NEW-TITLE-CODE                            PIC 999.
000740      05  NEW-EDUCATION-CODE                        PIC 9.
000750      05  NEW-PERFORMANCE-CODE                     PIC X.
000760      05  NEW-SALARY-DATA OCCURS 2 TIMES.
000770          10  NEW-SALARY                            PIC 9(5).
000780          10  NEW-SALARY-MONTH                       PIC 99.
000790          10  NEW-SALARY-YEAR                        PIC 99.
000800
000810 FD  DELETED-RECORD-FILE                            Record layout is identical to old master
000820      LABEL RECORDS ARE STANDARD
000830      RECORD CONTAINS 50 CHARACTERS
000840      DATA RECORD IS DELETED-MASTER-RECORD.
000850 01  DELETED-MASTER-RECORD                          PIC X(50).
000860
000870 WORKING-STORAGE SECTION.
000880 01  PROGRAM-SWITCHES.
000890      05  WS-OLD-MAST-READ-SWITCH                     PIC X(3)    VALUE SPACES.
000900      05  WS-TRANS-READ-SWITCH                       PIC X(3)    VALUE SPACES.
000910
000920 PROCEDURE DIVISION.
000930 010-UPDATE-MASTER-FILE.
000940      OPEN INPUT TRANSACTION-FILE
000950          OLD-MASTER-FILE
000960          OUTPUT NEW-MASTER-FILE
000970          DELETED-RECORD-FILE.
000980
000990      PERFORM 180-READ-OLD-MASTER-FILE.
001000      PERFORM 190-READ-TRANSACTION-FILE.
001010
001020      PERFORM 020-COMPARE-IDS
001030          UNTIL TRANS-SOC-SEC-NUMBER = HIGH-VALUES
001040          AND OLD-SOC-SEC-NUMBER = HIGH-VALUES.
001050
001060      CLOSE TRANSACTION-FILE
001070          OLD-MASTER-FILE
001080          NEW-MASTER-FILE
001090          DELETED-RECORD-FILE.
001100
001110      STOP RUN.
001120
001130 020-COMPARE-IDS.

```

FIGURE 11.6 *Continued*

Displays messages for testing

```

001140 DISPLAY " ".
001150 DISPLAY "RECORDS BEING PROCESSED".
001160 DISPLAY " TRANSACTION SOC SEC: " TRANS-SOC-SEC-NUMBER
001170 " TRANSACTION CODE: " TRANS-CODE.
001180 DISPLAY " OLD MASTER SOC SEC: " OLD-SOC-SEC-NUMBER.
001190
001200 IF OLD-SOC-SEC-NUMBER < TRANS-SOC-SEC-NUMBER
001210 PERFORM 050-COPY-OLD-RECORD
001220 ELSE
001230 IF OLD-SOC-SEC-NUMBER = TRANS-SOC-SEC-NUMBER
001240 PERFORM 060-PROCESS-MATCHING-IDS
001250 ELSE
001260 PERFORM 100-CHECK-MISSING-TRANSACTION.
001270
001280 IF WS-TRANS-READ-SWITCH = "YES"
001290 MOVE "NO " TO WS-TRANS-READ-SWITCH
001300 PERFORM 190-READ-TRANSACTION-FILE.
001310
001320 IF WS-OLD-MAST-READ-SWITCH = "YES"
001330 MOVE "NO " TO WS-OLD-MAST-READ-SWITCH
001340 PERFORM 180-READ-OLD-MASTER-FILE.
001350
001360 050-COPY-OLD-RECORD.
001370 DISPLAY "050-COPY-OLD-RECORD ROUTINE ENTERED".
001380 MOVE "YES" TO WS-OLD-MAST-READ-SWITCH.
001390
001400 060-PROCESS-MATCHING-IDS.
001410 DISPLAY "060-PROCESS-MATCHING-IDS ROUTINE ENTERED".
001420
001430 IF ADDITION
001440 PERFORM 070-WRITE-DUPLICATE-ADD-ERROR
001450 ELSE
001460 IF SALARY-UPDATE
001470 PERFORM 075-UPDATE-EMPLOYEE-SALARY
001480 ELSE
001490 IF CORRECTION
001500 PERFORM 080-CORRECT-OLD-RECORD
001510 ELSE
001520 IF DELETION
001530 PERFORM 090-DELETE-OLD-RECORD.
001540
001550 MOVE "YES" TO WS-TRANS-READ-SWITCH.
001560 MOVE "YES" TO WS-OLD-MAST-READ-SWITCH.
001570
001580 070-WRITE-DUPLICATE-ADD-ERROR.
001590 DISPLAY "070-WRITE-DUPLICATE-ADD-ERROR ROUTINE ENTERED".
001600
001610 075-UPDATE-EMPLOYEE-SALARY.
001620 DISPLAY "075-UPDATE-EMPLOYEE-SALARY ROUTINE ENTERED".
001630
001640 080-CORRECT-OLD-RECORD.
001650 DISPLAY "080-CORRECT-OLD-RECORD ROUTINE ENTERED".
001660
001670 090-DELETE-OLD-RECORD.
001680 DISPLAY "090-DELETE-OLD-RECORD ROUTINE ENTERED".
001690
001700 100-CHECK-MISSING-TRANSACTION.
001710 DISPLAY "100-CHECK-MISSING-TRANSACTION ROUTINE ENTERED".
001720 IF ADDITION
001730 DISPLAY "RECORD WILL BE ADDED " TRANS-SOC-SEC-NUMBER
001740 ELSE
001750 DISPLAY "ERROR - NO CORRESPONDING MASTER RECORD FOR "
001760 TRANS-SOC-SEC-NUMBER.
001770
001780 MOVE "YES" TO WS-TRANS-READ-SWITCH.
001790
001800
001810 180-READ-OLD-MASTER-FILE.
001820 READ OLD-MASTER-FILE
001830 AT END MOVE HIGH-VALUES TO OLD-SOC-SEC-NUMBER.
001840
001850 190-READ-TRANSACTION-FILE.
001860 READ TRANSACTION-FILE
001870 AT END MOVE HIGH-VALUES TO TRANS-SOC-SEC-NUMBER.

```

Nested IF drives the program

Reads old master and/or transaction file

Program stubs

FIGURE 11.6 Continued

The Environment Division provides for four files as per the program specifications. The Data Division is straightforward and consists essentially of FDs for the aforementioned files. The Procedure Division embodies the essential logic of the pseudocode of Figure 11.4. Figure 11.7 contains both test data and corresponding output, which was produced by the “stubs” pro-

```

11111111SMITH          ATL0105A180000781000000000
22222222JONES          CHI0404E230000381200000980
33333333BAKER          BOS0306P195000182180001280
44444444MILGROM        DET0207G280000481240000480
55555555CRAWFORD       WAS0305E300000581250001179

a) Old Master file

10000000BLACKBURN      CHI                                C
22222222                250000981                        U
33333333BAKER          DET      E      1281                C
40000000WICKS          CHI0206A350000781                    A
44444444MILGROM        DET0207G280000481240000480A
55555555CRAWFORD       55555555CRAWFORD                    D
77777777SAMUEL         ATL0407E300000881                    A

b) Transaction file

RECORDS BEING PROCESSED
  TRANSACTION SOC SEC: 100000000 TRANSACTION CODE: C
  OLD MASTER SOC SEC: 111111111
100-CHECK-MISSING-TRANSACTION ROUTINE ENTERED
ERROR - NO CORRESPONDING MASTER RECORD FOR 100000000

RECORDS BEING PROCESSED
  TRANSACTION SOC SEC: 222222222 TRANSACTION CODE: U
  OLD MASTER SOC SEC: 111111111
050-COPY-OLD-RECORD ROUTINE ENTERED

RECORDS BEING PROCESSED
  TRANSACTION SOC SEC: 222222222 TRANSACTION CODE: U
  OLD MASTER SOC SEC: 222222222
060-PROCESS-MATCHING-IDS ROUTINE ENTERED
075-UPDATE-EMPLOYEE-SALARY ROUTINE ENTERED

RECORDS BEING PROCESSED
  TRANSACTION SOC SEC: 333333333 TRANSACTION CODE: C
  OLD MASTER SOC SEC: 333333333
060-PROCESS-MATCHING-IDS ROUTINE ENTERED
080-CORRECT-OLD-RECORD ROUTINE ENTERED

RECORDS BEING PROCESSED
  TRANSACTION SOC SEC: 400000000 TRANSACTION CODE: A
  OLD MASTER SOC SEC: 444444444
100-CHECK-MISSING-TRANSACTION ROUTINE ENTERED
RECORD WILL BE ADDED 400000000

RECORDS BEING PROCESSED
  TRANSACTION SOC SEC: 444444444 TRANSACTION CODE: A
  OLD MASTER SOC SEC: 444444444
060-PROCESS-MATCHING-IDS ROUTINE ENTERED
070-WRITE-DUPLICATE-ADD-ERROR ROUTINE ENTERED

RECORDS BEING PROCESSED
  TRANSACTION SOC SEC: 555555555 TRANSACTION CODE: D
  OLD MASTER SOC SEC: 555555555
060-PROCESS-MATCHING-IDS ROUTINE ENTERED
090-DELETE-OLD-RECORD ROUTINE ENTERED

RECORDS BEING PROCESSED
  TRANSACTION SOC SEC: 777777777 TRANSACTION CODE: A
  OLD MASTER SOC SEC:
100-CHECK-MISSING-TRANSACTION ROUTINE ENTERED
RECORD WILL BE ADDED 777777777

```

FIGURE 11.7 Test data and stubs output. (a) Old master file. (b) Transaction file. (c) Stubs output.

gram of Figure 11.6. A critical analysis of that output will confirm that the program appears to work correctly.

The Procedure Division begins by reading the first record from each file, social security numbers 100000000 and 111111111, for the transaction and old master files respectively. The current transaction is *not* in the old master, yet the transaction code was a C. The routine CHECK-MISSING-TRANSACTION was executed and an error message produced.

The program retains the old master record with social security number 111111111, but reads a new transaction record, social security number 222222222 and transaction code U. The COPY-OLD-RECORD paragraph is executed correctly, indicating no activity for the old master record, social security number 111111111.

This time the program retains the current transaction record, social security number 222222222, but reads a new old master record. The social security numbers match (both are 222222222) causing execution of PROCESS-MATCHING-IDS, followed by UPDATE-EMPLOYEE-SALARY.

New records are taken from *both* the old master and transaction files, with social security numbers again matching (both are 333333333). This time, however, CORRECT-OLD-RECORD is executed after PROCESS-MATCHING-IDS, because of the C transaction code.

The reader may begin to appreciate that the program appears to be functioning correctly. *Paragraphs are executed in the correct order.* Note well how records are appropriately read from the two input files; i.e., sometimes from the transaction file only, sometimes from the old master only, and sometimes from both. Realize also that virtually no processing is done in the lowest level paragraphs; e.g., CORRECT-OLD-RECORD, UPDATE-EMPLOYEE-SALARY, and so on. Nevertheless, the testing is worthwhile because it ensures that the higher level modules are working as intended. It is a relatively trivial matter to complete the program stubs and produce the completed program of Figure 11.8.

The remainder of Figure 11.7 is left to the reader as an exercise.

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID.    SEQUUPDAT.
000120 AUTHOR.        R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER.    TRS-80.
000170 OBJECT-COMPUTER.    TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210     SELECT TRANSACTION-FILE
000220         ASSIGN TO INPUT "TRANS/DAT".
000230     SELECT OLD-MASTER-FILE
000240         ASSIGN TO INPUT "TABLES/DAT".
000250     SELECT NEW-MASTER-FILE
000260         ASSIGN TO OUTPUT "NEWMAS/DAT".
000270     SELECT DELETED-RECORD-FILE
000280         ASSIGN TO OUTPUT "DELETED/DAT".
000290
000300 DATA DIVISION.
000310 FILE SECTION.
000320 FD  TRANSACTION-FILE
000330     LABEL RECORDS ARE STANDARD
000340     RECORD CONTAINS 51 CHARACTERS
000350     DATA RECORD IS TRANS-RECORD.

```

Four files are required

FIGURE 11.8 Completed sequential update

```

000360 01 TRANS-RECORD.
000370 05 TRANS-SOC-SEC-NUMBER PIC X(9).
000380 05 TRANS-LAST-NAME PIC X(15).
000390 05 TRANS-LOC-CODE PIC X(3).
000400 05 TRANS-TITLE-CODE PIC 999.
000410 05 TRANS-EDUCATION-CODE PIC 9.
000420 05 TRANS-PERFORMANCE-CODE PIC X.
000430 05 TRANS-SALARY PIC 9(5).
000440 05 TRANS-SALARY-MONTH PIC 99.
000450 05 TRANS-SALARY-YEAR PIC 99.
000460 05 FILLER PIC X(9).
000470 05 TRANS-CODE PIC X.
000471 88 ADDITION VALUE "A".
000472 88 CORRECTION VALUE "C".
000473 88 DELETION VALUE "D".
000474 88 SALARY-UPDATE VALUE "U".
000480
000490 FD OLD-MASTER-FILE Four types of transactions
000500 LABEL RECORDS ARE STANDARD
000510 RECORD CONTAINS 50 CHARACTERS
000520 DATA RECORD IS OLD-MASTER-RECORD.
000530 01 OLD-MASTER-RECORD.
000540 05 OLD-SOC-SEC-NUMBER PIC X(9).
000550 05 OLD-LAST-NAME PIC X(15).
000560 05 OLD-LOC-CODE PIC X(3).
000570 05 OLD-TITLE-CODE PIC 999.
000580 05 OLD-EDUCATION-CODE PIC 9.
000590 05 OLD-PERFORMANCE-CODE PIC X.
000600 05 OLD-SALARY-DATA OCCURS 2 TIMES.
000610 10 OLD-SALARY PIC 9(5).
000620 10 OLD-SALARY-MONTH PIC 99.
000630 10 OLD-SALARY-YEAR PIC 99.
000640
000650 FD NEW-MASTER-FILE Two levels of salary data
000660 LABEL RECORDS ARE STANDARD
000670 RECORD CONTAINS 50 CHARACTERS
000680 DATA RECORD IS NEW-MASTER-RECORD.
000690 01 NEW-MASTER-RECORD.
000700 05 NEW-SOC-SEC-NUMBER PIC X(9).
000710 05 NEW-LAST-NAME PIC X(15).
000720 05 NEW-LOC-CODE PIC X(3).
000730 05 NEW-TITLE-CODE PIC 999.
000740 05 NEW-EDUCATION-CODE PIC 9.
000750 05 NEW-PERFORMANCE-CODE PIC X.
000760 05 NEW-SALARY-DATA OCCURS 2 TIMES.
000770 10 NEW-SALARY PIC 9(5).
000780 10 NEW-SALARY-MONTH PIC 99.
000790 10 NEW-SALARY-YEAR PIC 99.
000800
000810 FD DELETED-RECORD-FILE Record layout identical to old master
000820 LABEL RECORDS ARE STANDARD
000830 RECORD CONTAINS 50 CHARACTERS
000840 DATA RECORD IS DELETED-MASTER-RECORD.
000850 01 DELETED-MASTER-RECORD PIC X(50).
000860
000870 WORKING-STORAGE SECTION.
000880 01 PROGRAM-SWITCHES.
000890 05 WS-OLD-MAST-READ-SWITCH PIC X(3) VALUE SPACES.
000900 05 WS-TRANS-READ-SWITCH PIC X(3) VALUE SPACES.
000910
000920 PROCEDURE DIVISION.
000930 010-UPDATE-MASTER-FILE.
000940 OPEN INPUT TRANSACTION-FILE
000950 OLD-MASTER-FILE
000960 OUTPUT NEW-MASTER-FILE
000970 DELETED-RECORD-FILE. Initial reads
000980
000990 PERFORM 180-READ-OLD-MASTER-FILE.
001000 PERFORM 190-READ-TRANSACTION-FILE.
001010

```

FIGURE 11.8 Continued


```

001020    PERFORM 020-COMPARE-IDS
001030        UNTIL TRANS-SOC-SEC-NUMBER = HIGH-VALUES
001040        AND OLD-SOC-SEC-NUMBER = HIGH-VALUES.
001050
001060    CLOSE TRANSACTION-FILE
001070        OLD-MASTER-FILE
001080        NEW-MASTER-FILE
001090        DELETED-RECORD-FILE.
001100
001110    STOP RUN.
001120
001130 020-COMPARE-IDS.
001200    IF OLD-SOC-SEC-NUMBER < TRANS-SOC-SEC-NUMBER
001210        PERFORM 050-COPY-OLD-RECORD
001220    ELSE
001230        IF OLD-SOC-SEC-NUMBER = TRANS-SOC-SEC-NUMBER
001240            PERFORM 060-PROCESS-MATCHING-IDS
001250        ELSE
001260            PERFORM 100-CHECK-MISSING-TRANSACTION.
001270
001280    IF WS-TRANS-READ-SWITCH = "YES"
001290        MOVE "NO " TO WS-TRANS-READ-SWITCH
001300        PERFORM 190-READ-TRANSACTION-FILE.
001310
001320    IF WS-OLD-MAST-READ-SWITCH = "YES"
001330        MOVE "NO " TO WS-OLD-MAST-READ-SWITCH
001340        PERFORM 180-READ-OLD-MASTER-FILE.
001350
001360 050-COPY-OLD-RECORD.
001370    MOVE OLD-MASTER-RECORD TO NEW-MASTER-RECORD.
001371    WRITE NEW-MASTER-RECORD.
001380    MOVE "YES" TO WS-OLD-MAST-READ-SWITCH.
001390
001400 060-PROCESS-MATCHING-IDS.
001430    IF ADDITION
001440        PERFORM 070-WRITE-DUPLICATE-ADD-ERROR
001450    ELSE
001460        IF SALARY-UPDATE
001470            PERFORM 075-UPDATE-EMPLOYEE-SALARY
001480        ELSE
001490            IF CORRECTION
001500                PERFORM 080-CORRECT-OLD-RECORD
001510            ELSE
001520                IF DELETION
001530                    PERFORM 090-DELETE-OLD-RECORD.
001540
001550    MOVE "YES" TO WS-TRANS-READ-SWITCH.
001560    MOVE "YES" TO WS-OLD-MAST-READ-SWITCH.
001570
001580 070-WRITE-DUPLICATE-ADD-ERROR.
001590    DISPLAY "ERROR - RECORD ALREADY EXISTS IN MASTER FILE: "
001591        TRANS-SOC-SEC-NUMBER.
001592    MOVE OLD-MASTER-RECORD TO NEW-MASTER-RECORD.
001593    WRITE NEW-MASTER-RECORD.
001600
001610 075-UPDATE-EMPLOYEE-SALARY.
001620    MOVE OLD-MASTER-RECORD TO NEW-MASTER-RECORD.
001621    IF NEW-SALARY (1) = TRANS-SALARY
001622        DISPLAY "ERROR - SALARY UPDATE ALREADY DONE: "
001623            TRANS-SOC-SEC-NUMBER
001624    ELSE
001625        MOVE NEW-SALARY-DATA (1) TO NEW-SALARY-DATA (2)
001626        MOVE TRANS-SALARY TO NEW-SALARY (1)
001627        MOVE TRANS-SALARY-MONTH TO NEW-SALARY-MONTH (1)
001628        MOVE TRANS-SALARY-YEAR TO NEW-SALARY-YEAR (1).

```

Nested IF statement drives the program

Determines additional processing for matching transaction

Check for duplicate update

Present data is moved to previous data

FIGURE 11.8 *Continued*

```

001629
001630 WRITE NEW-MASTER-RECORD.
001631
001640 080-CORRECT-OLD-RECORD.
001641 MOVE OLD-MASTER-RECORD TO NEW-MASTER-RECORD.
001642 IF TRANS-LAST-NAME NOT = SPACES
001643     MOVE TRANS-LAST-NAME TO NEW-LAST-NAME.
001644 IF TRANS-LOC-CODE NOT = SPACES
001645     MOVE TRANS-LOC-CODE TO NEW-LOC-CODE.
001646 IF TRANS-TITLE-CODE NOT = SPACES
001647     MOVE TRANS-TITLE-CODE TO NEW-TITLE-CODE.
001648 IF TRANS-EDUCATION-CODE NOT = SPACES
001649     MOVE TRANS-EDUCATION-CODE TO NEW-EDUCATION-CODE.
001650 IF TRANS-PERFORMANCE-CODE NOT = SPACES
001651     MOVE TRANS-PERFORMANCE-CODE TO NEW-PERFORMANCE-CODE.
001652 IF TRANS-SALARY NOT = SPACES
001653     MOVE TRANS-SALARY TO NEW-SALARY (1).
001654 IF TRANS-SALARY-MONTH NOT = SPACES
001655     MOVE TRANS-SALARY-MONTH TO NEW-SALARY-MONTH (1).
001656 IF TRANS-SALARY-YEAR NOT = SPACES
001657     MOVE TRANS-SALARY-YEAR TO NEW-SALARY-YEAR (1).
001658
001659 WRITE NEW-MASTER-RECORD.
001660
001670 090-DELETE-OLD-RECORD.
001680 MOVE OLD-MASTER-RECORD TO DELETED-MASTER-RECORD.
001681 WRITE DELETED-MASTER-RECORD.
001690
001700 100-CHECK-MISSING-TRANSACTION.
001720 IF ADDITION
001721     MOVE TRANS-SOC-SEC-NUMBER TO NEW-SOC-SEC-NUMBER
001722     MOVE TRANS-LAST-NAME TO NEW-LAST-NAME
001723     MOVE TRANS-LOC-CODE TO NEW-LOC-CODE
001724     MOVE TRANS-TITLE-CODE TO NEW-TITLE-CODE
001725     MOVE TRANS-EDUCATION-CODE TO NEW-EDUCATION-CODE
001726     MOVE TRANS-PERFORMANCE-CODE TO NEW-PERFORMANCE-CODE
001727     MOVE TRANS-SALARY TO NEW-SALARY (1)
001728     MOVE TRANS-SALARY-MONTH TO NEW-SALARY-MONTH (1)
001729     MOVE TRANS-SALARY-YEAR TO NEW-SALARY-YEAR (1)
001730     MOVE ZEROS TO NEW-SALARY-DATA (2)
001731     WRITE NEW-MASTER-RECORD
001740 ELSE
001750     DISPLAY "ERROR - NO CORRESPONDING MASTER RECORD FOR "
001760         TRANS-SOC-SEC-NUMBER.
001770
001780 MOVE "YES" TO WS-TRANS-READ-SWITCH.
001790
001800
001810 180-READ-OLD-MASTER-FILE.
001820 READ OLD-MASTER-FILE
001830     AT END MOVE HIGH-VALUES TO OLD-SOC-SEC-NUMBER.
001840
001850 190-READ-TRANSACTION-FILE.
001860 READ TRANSACTION-FILE
001870     AT END MOVE HIGH-VALUES TO TRANS-SOC-SEC-NUMBER.

```

Expanded from a program stub

Zeros out previous salary

FIGURE 11.8 Continued

THE COMPLETED PROGRAM

Once the skeletal maintenance program has been tested and debugged, it is relatively simple to "fill in the blanks" and complete the maintenance program. In other words, the most difficult portion of the maintenance program is the *interaction* between modules, that is, whether to read from the old master or the transaction file, or both; and which lower level module to call, that is, CORRECT-OLD-RECORD, UPDATE-EMPLOYEE-SALARY, and so on. The details of these modules are easy by comparison.

As is to be expected, Figure 11.8 is longer than its predecessor, due to the expansion of the program stubs. The essential structure of the Procedure

Division is unchanged, however, and the nested IF of lines 1200-1260 still drives the program.

The logic of the lower level modules is straightforward and easy to follow. The UPDATE-EMPLOYEE-SALARY module first verifies that the salary update has not been done. It then moves what had been the present salary data (SALARY-DATA (1)) to the previous slot (SALARY-DATA (2)), and moves the transaction data to the present salary.

The CORRECT-OLD-RECORD routine moves the current old master record to the new master, then checks the transaction fields one at a time for changes. Any time a field is to be changed, i.e., the transaction field is not blank, the transaction data is moved to the new master. Finally, the new master record is written.

DELETE-OLD-RECORD writes the current old master record to a special file, DELETED-RECORD-FILE, for possible recall at a future date. It deletes the record from the new master file, simply by *not* writing it.

11111111SMITH	ATL0105A180000781000000000
22222222JONES	CHI0404E230000381200000980
33333333BAKER	BOS0306P195000182180001280
44444444MILGROM	DET0207G280000481240000480
55555555CRAWFORD	WAS0305E300000581250001179

Record will be deleted (see Transaction File)

a) Old Master

10000000BLACKBURN	CHI		C
22222222		250000981	U
33333333BAKER	DET	E 1281	C
40000000WICKS	CHI0206A350000781		A
44444444MILGROM	DET0207G280000481240000480		A
55555555CRAWFORD			D
77777777SAMUEL	ATL0407E300000881		A

b) Transactions

11111111SMITH	ATL0105A180000781000000000	
22222222JONES	CHI0404E250000981230000381	Location was corrected
33333333BAKER	DET0306E195001281180001280	Salary update has been done
40000000WICKS	CHI0206A350000781000000000	
44444444MILGROM	DET0207G280000481240000480	
77777777SAMUEL	ATL0407E300000881000000000	

Record has been added

c) New Master

ERROR - NO CORRESPONDING MASTER RECORD FOR 10000000
 ERROR - RECORD ALREADY EXISTS IN MASTER FILE: 44444444

d) Error Messages

55555555CRAWFORD	WAS0305E300000581250001179
------------------	----------------------------

e) Deleted File

FIGURE 11.9 Test data and output for sequential update. (a) Old master. (b) Transactions. (c) New master. (d) Error messages. (e) Deleted file.

Finally, CHECK-MISSING-TRANSACTION is executed when the current transaction record is not present in the old master. If the transaction code is an A, the record is written to the new master file; if the code is not an A, the transaction record is flagged as a “no match.”

Figure 11.9 contains both input and output associated with the program of Figure 11.8. Observe that the incoming test data are identical to the input to the stubs program (Figure 11.7), but that the output is significantly different.

SUMMARY This chapter is one of the most significant in the entire book. Although printed reports (or terminal displays) are a more tangible result of data processing, the files on which these reports are based must be periodically changed to reflect the changing nature of the physical environment. Hence the importance of file maintenance.

The chapter began with a conceptual discussion of file maintenance, the need for backup and recovery procedures, and the importance of error checking. The chapter referred to the concept of a “stand alone” edit program in which the transaction file is processed separately *prior* to the update. The intent of such a program is to “scrub the data”, and ensure that only valid transactions are presented to the update program. The following is a list of typical error checks:

1. **Numeric test:** A numeric test ensures that a COBOL-specified numeric field, that is, one defined with a picture of 9's, does in fact contain numeric data. Note well that commas, decimal points, or blanks are not numeric and will cause problems in execution. The test is quite easy to implement in COBOL and takes the form

```
IF EMPLOYEE-SALARY IS NOT NUMERIC
  PERFORM ERROR-ROUTINE.
```

2. **Alphabetic test:** Analogous to a numeric test, except that it checks that alphabetic fields have been coded. Any errors detected here are typically less serious than for numeric fields.
3. **Reasonableness check:** A reasonableness check assures that a given number is within “normal” bounds. These tests often take the form of *limit* checks, that is, testing that a value does not exceed a designated upper or lower extreme. For example, a payroll program may check that no hourly worker’s pay exceeds \$400 per week. A weekly gross that exceeded this amount might be deemed unreasonable and the transaction flagged for further scrutiny. A *range* check, which ensures that a given value is within specified limits, is another form of reasonableness check.
4. **Consistency check:** Verifies that the values in two or more fields are consistent, for example, salary and title. Since salary is partly a function of title, there should be a correlation between the two. Another example of a consistency check would be an individual’s credit rating and the amount of credit a bank is willing to extend.

5. **Checking that a code exists:** One of the most common, yet most important tests. This author has seen countless errors compounded because this check was not implemented. For example:

```
IF SEX = "M"
  ADD 1 TO NUMBER-OF-MEN
ELSE
  ADD 1 TO NUMBER-OF-WOMEN.
```

It is decidedly *poor* practice to assume that an incoming record is female if it is not male. Rather, both codes should be explicitly checked, and if neither occurs, a suitable error message should be printed.

6. **Sequence check:** This assures that incoming records are in proper order. It can also be used when several lines (in a data file) comprise one record to assure that the lines within a record are in proper sequence. (The latter is described further under completeness check.)
7. **Completeness check:** Verifies that all fields in a record, e.g., an addition, are present. Realize, however, that all discussions to date have assumed a one-line record. It is also possible to have several lines (in a data file) in sequence comprise a single record. In this instance, the completeness check would ensure that the requisite number of lines were present for each record.
8. **Date check:** Ensures that an incoming date is acceptable. There are several variations in the way this test can be performed. For example, birth dates can be checked to ensure that no one would be hired who was younger than 16 or older than 65. Checks made on a date can also test that the day falls between 1 and 31, the month between 1 and 12, and the year within a designated period, often just the current year.

Realize, however, that even with such a complete edit program, there are certain errors which can only be detected in the maintenance program itself. These include miscopied social security numbers (i.e., no matches) and duplicate additions. The chapter developed a file maintenance program and included these error checks.

Considerable space was also devoted to the concept of top down development and testing; i.e., testing a program before it is completely finished. The concepts of program stubs and hierarchy charts were covered in this discussion.

TRUE/FALSE

1. A program stub may be a one-sentence paragraph containing a DISPLAY statement.
2. A COBOL program must be fully coded before testing can begin.
3. The lower level modules in a hierarchy chart should be tested first.
4. The modules in a hierarchy chart should be dependent on one another.
5. Pseudocode has precise syntactical rules.
6. A box in a hierarchy chart should depict more than one function for efficiency.
7. A flowchart and a hierarchy chart convey similar kinds of information.

8. A sequential update requires that every record be copied to the new master even if it didn't change.
9. A sequential update requires both the old and new master files to be in sequence.
10. A sequential update has two distinct master files; an old and a new.

EXERCISES

1. Given the old master file of Figure 11.9a, and the following transactions:

100000000CALDWELL	CHI0206A350000981	A
111111111SMITH	DET020	C
333333333BAKER		D
555555555CRAWFORD	350000981	U
666666666SHERRY	CHI	C

Indicate the intended new master, error messages, and deleted file.

2. *Debugging*: Figure 11.10 contains *invalid* output produced by the *incorrect* maintenance program of Figure 11.11. (The latter is a modified version of the program of Figure 11.8.) Find and correct the errors in Figure 11.10 so that it produces correct output; i.e., results identical to Figure 11.9 in the text.

100000000BLACKBURN	CHI		C
222222222		250000981	U
333333333BAKER	DET	E 1281	C
400000000WICKS	CHI0206A350000781		A
444444444MILGROM	DET0207G280000481240000480A		
555555555CRAWFORD			D
777777777SAMUEL	ATL0407E300000881		A

Record missing in new master
Record was not added
a) Transaction File

111111111SMITH	ATL0105A180000781000000000
222222222JONES	CHI0404E230000381200000980
333333333BAKER	BOS0306P195000182180001280
444444444MILGROM	DET0207G280000481240000480
555555555CRAWFORD	WAS0305E300000581250001179

Record was unintentionally deleted
b) Old Master

111111111SMITH	ATL0105A180000781000000000
222222222JONES	CHI0404E250000981250000981
400000000WICKS	CHI0306A350000781000000000
444444444MILGROM	DET0207G280000481240000480

Salary update did not work
Title is wrong on this record
c) New Master

555555555CRAWFORD	WAS0305E300000581250001179
-------------------	----------------------------

d) Deleted File

ERROR - NO CORRESPONDING MASTER RECORD FOR 100000000
ERROR - RECORD ALREADY EXISTS IN MASTER FILE: 444444444

e) Error Messages

FIGURE 11.10 Invalid output from sequential update, (a) Transaction file, (b) Old master, (c) New master, (d) Deleted file, (e) Error messages.

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID. ESEQUPDA.
000120 AUTHOR. R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER. TRS-80.
000170 OBJECT-COMPUTER. TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210 SELECT TRANSACTION-FILE
000220 ASSIGN TO INPUT "TRANS/DAT".
000230 SELECT OLD-MASTER-FILE
000240 ASSIGN TO INPUT "TABLES/DAT".
000250 SELECT NEW-MASTER-FILE
000260 ASSIGN TO OUTPUT "NEWMAS/DAT".
000270 SELECT DELETED-RECORD-FILE
000280 ASSIGN TO OUTPUT "DELETED/DAT".
000290
000300 DATA DIVISION.
000310 FILE SECTION.
000320 FD TRANSACTION-FILE
000330 LABEL RECORDS ARE STANDARD
000340 RECORD CONTAINS 51 CHARACTERS
000350 DATA RECORD IS TRANS-RECORD.
000360 01 TRANS-RECORD.
000370 05 TRANS-SOC-SEC-NUMBER PIC X(9).
000380 05 TRANS-LAST-NAME PIC X(15).
000390 05 TRANS-LOC-CODE PIC X(3).
000400 05 TRANS-TITLE-CODE PIC 999.
000410 05 TRANS-EDUCATION-CODE PIC 9.
000420 05 TRANS-PERFORMANCE-CODE PIC X.
000430 05 TRANS-SALARY PIC 9(5).
000440 05 TRANS-SALARY-MONTH PIC 99.
000450 05 TRANS-SALARY-YEAR PIC 99.
000460 05 FILLER PIC X(9).
000470 05 TRANS-CODE PIC X.
000471 88 ADDITION VALUE "A".
000472 88 CORRECTION VALUE "C".
000473 88 DELETION VALUE "D".
000474 88 SALARY-UPDATE VALUE "U".
000480
000490 FD OLD-MASTER-FILE
000500 LABEL RECORDS ARE STANDARD
000510 RECORD CONTAINS 50 CHARACTERS
000520 DATA RECORD IS OLD-MASTER-RECORD.
000530 01 OLD-MASTER-RECORD.
000540 05 OLD-SOC-SEC-NUMBER PIC X(9).
000550 05 OLD-LAST-NAME PIC X(15).
000560 05 OLD-LOC-CODE PIC X(3).
000570 05 OLD-TITLE-CODE PIC 999.
000580 05 OLD-EDUCATION-CODE PIC 9.
000590 05 OLD-PERFORMANCE-CODE PIC X.
000600 05 OLD-SALARY-DATA OCCURS 2 TIMES.
000610 10 OLD-SALARY PIC 9(5).
000620 10 OLD-SALARY-MONTH PIC 99.
000630 10 OLD-SALARY-YEAR PIC 99.
000640
000650 FD NEW-MASTER-FILE
000660 LABEL RECORDS ARE STANDARD
000670 RECORD CONTAINS 50 CHARACTERS
000680 DATA RECORD IS NEW-MASTER-RECORD.
000690 01 NEW-MASTER-RECORD.
000700 05 NEW-SOC-SEC-NUMBER PIC X(9).
000710 05 NEW-LAST-NAME PIC X(15).
000720 05 NEW-LOC-CODE PIC X(3).
000730 05 NEW-TITLE-CODE PIC 999.
000740 05 NEW-EDUCATION-CODE PIC 9.
000750 05 NEW-PERFORMANCE-CODE PIC X.
000760 05 NEW-SALARY-DATA OCCURS 2 TIMES.
000770 10 NEW-SALARY PIC 9(5).

```

FIGURE 11.11 Incorrect sequential maintenance program

```

000780          10 NEW-SALARY-MONTH          PIC 99.
000790          10 NEW-SALARY-YEAR          PIC 99.
000800
000810 FD DELETED-RECORD-FILE
000820 LABEL RECORDS ARE STANDARD
000830 RECORD CONTAINS 50 CHARACTERS
000840 DATA RECORD IS DELETED-MASTER-RECORD.
000850 01 DELETED-MASTER-RECORD          PIC X(50).
000860
000870 WORKING-STORAGE SECTION.
000880 01 PROGRAM-SWITCHES.
000890    05 WS-OLD-MAST-READ-SWITCH          PIC X(3)    VALUE SPACES.
000900    05 WS-TRANS-READ-SWITCH            PIC X(3)    VALUE SPACES.
000910
000920 PROCEDURE DIVISION.
000930 010-UPDATE-MASTER-FILE.
000940     OPEN INPUT TRANSACTION-FILE
000950         OLD-MASTER-FILE
000960         OUTPUT NEW-MASTER-FILE
000970         DELETED-RECORD-FILE.
000980
000990     PERFORM 180-READ-OLD-MASTER-FILE.
001000     PERFORM 190-READ-TRANSACTION-FILE.
001010
001020     PERFORM 020-COMPARE-IDS
001030         UNTIL TRANS-SOC-SEC-NUMBER = HIGH-VALUES
001040         OR OLD-SOC-SEC-NUMBER = HIGH-VALUES.
001050
001060     CLOSE TRANSACTION-FILE
001070         OLD-MASTER-FILE
001080         NEW-MASTER-FILE
001090         DELETED-RECORD-FILE.
001100
001110     STOP RUN.
001120
001130 020-COMPARE-IDS.
001200     IF OLD-SOC-SEC-NUMBER < TRANS-SOC-SEC-NUMBER
001210         PERFORM 050-COPY-OLD-RECORD
001220     ELSE
001230         IF OLD-SOC-SEC-NUMBER = TRANS-SOC-SEC-NUMBER
001240             PERFORM 060-PROCESS-MATCHING-IDS
001250         ELSE
001260             PERFORM 100-CHECK-MISSING-TRANSACTION.
001270
001280     IF WS-TRANS-READ-SWITCH = "YES"
001290         MOVE "NO " TO WS-TRANS-READ-SWITCH
001300         PERFORM 190-READ-TRANSACTION-FILE.
001310
001320     IF WS-OLD-MAST-READ-SWITCH = "YES"
001330         MOVE "NO " TO WS-OLD-MAST-READ-SWITCH
001340         PERFORM 180-READ-OLD-MASTER-FILE.
001350
001360 050-COPY-OLD-RECORD.
001370     MOVE OLD-MASTER-RECORD TO NEW-MASTER-RECORD.
001371     WRITE NEW-MASTER-RECORD.
001380     MOVE "YES" TO WS-OLD-MAST-READ-SWITCH.
001390
001400 060-PROCESS-MATCHING-IDS.
001430     IF ADDITION
001440         PERFORM 070-WRITE-DUPLICATE-ADD-ERROR
001450     ELSE
001460         IF SALARY-UPDATE
001470             PERFORM 075-UPDATE-EMPLOYEE-SALARY
001480         ELSE
001490             IF CORRECTION
001500                 PERFORM 080-CORRECT-OLD-RECORD
001510             ELSE
001520                 IF DELETION
001530                     PERFORM 090-DELETE-OLD-RECORD.
001540
001550     MOVE "YES" TO WS-TRANS-READ-SWITCH.
001560     MOVE "YES" TO WS-OLD-MAST-READ-SWITCH.

```

FIGURE 11.11 *Continued*


```

001570
001580 070-WRITE-DUPLICATE-ADD-ERROR.
001590     DISPLAY "ERROR - RECORD ALREADY EXISTS IN MASTER FILE: "
001591         TRANS-SOC-SEC-NUMBER.
001592     MOVE OLD-MASTER-RECORD TO NEW-MASTER-RECORD.
001593     WRITE NEW-MASTER-RECORD.
001600
001610 075-UPDATE-EMPLOYEE-SALARY.
001620     MOVE OLD-MASTER-RECORD TO NEW-MASTER-RECORD.
001621     IF NEW-SALARY (1) = TRANS-SALARY
001622         DISPLAY "ERROR - SALARY UPDATE ALREADY DONE: "
001623             TRANS-SOC-SEC-NUMBER
001624     ELSE
001625         MOVE TRANS-SALARY TO NEW-SALARY (1)
001626         MOVE TRANS-SALARY-MONTH TO NEW-SALARY-MONTH (1)
001627         MOVE TRANS-SALARY-YEAR TO NEW-SALARY-YEAR (1)
001628         MOVE NEW-SALARY-DATA (1) TO NEW-SALARY-DATA (2).
001629
001630     WRITE NEW-MASTER-RECORD.
001631
001640 080-CORRECT-OLD-RECORD.
001641     MOVE OLD-MASTER-RECORD TO NEW-MASTER-RECORD.
001642     IF TRANS-LAST-NAME NOT = SPACES
001643         MOVE TRANS-LAST-NAME TO NEW-LAST-NAME.
001644     IF TRANS-LOC-CODE NOT = SPACES
001645         MOVE TRANS-LOC-CODE TO NEW-LOC-CODE.
001646     IF TRANS-TITLE-CODE NOT = SPACES
001647         MOVE TRANS-TITLE-CODE TO NEW-TITLE-CODE.
001648     IF TRANS-EDUCATION-CODE NOT = SPACES
001649         MOVE TRANS-EDUCATION-CODE TO NEW-EDUCATION-CODE.
001650     IF TRANS-PERFORMANCE-CODE NOT = SPACES
001651         MOVE TRANS-PERFORMANCE-CODE TO NEW-PERFORMANCE-CODE.
001652     IF TRANS-SALARY NOT = SPACES
001653         MOVE TRANS-SALARY TO NEW-SALARY (1).
001654     IF TRANS-SALARY-MONTH NOT = SPACES
001655         MOVE TRANS-SALARY-MONTH TO NEW-SALARY-MONTH (1).
001656     IF TRANS-SALARY-YEAR NOT = SPACES
001657         MOVE TRANS-SALARY-YEAR TO NEW-SALARY-YEAR (1).
001658
001670 090-DELETE-OLD-RECORD.
001680     MOVE OLD-MASTER-RECORD TO DELETED-MASTER-RECORD.
001681     WRITE DELETED-MASTER-RECORD.
001690
001700 100-CHECK-MISSING-TRANSACTION.
001720     IF ADDITION
001721         MOVE TRANS-SOC-SEC-NUMBER TO NEW-SOC-SEC-NUMBER
001722         MOVE TRANS-LAST-NAME TO NEW-LAST-NAME
001723         MOVE TRANS-LOC-CODE TO NEW-LOC-CODE
001725         MOVE TRANS-EDUCATION-CODE TO NEW-EDUCATION-CODE
001726         MOVE TRANS-PERFORMANCE-CODE TO NEW-PERFORMANCE-CODE
001727         MOVE TRANS-SALARY TO NEW-SALARY (1)
001728         MOVE TRANS-SALARY-MONTH TO NEW-SALARY-MONTH (1)
001729         MOVE TRANS-SALARY-YEAR TO NEW-SALARY-YEAR (1)
001730         MOVE ZEROS TO NEW-SALARY-DATA (2)
001731         WRITE NEW-MASTER-RECORD
001740     ELSE
001750         DISPLAY "ERROR - NO CORRESPONDING MASTER RECORD FOR "
001760             TRANS-SOC-SEC-NUMBER.
001770
001780     MOVE "YES" TO WS-TRANS-READ-SWITCH.
001790
001800
001810 180-READ-OLD-MASTER-FILE.
001820     READ OLD-MASTER-FILE
001830         AT END MOVE HIGH-VALUES TO OLD-SOC-SEC-NUMBER.
001840
001850 190-READ-TRANSACTION-FILE.
001860     READ TRANSACTION-FILE
001870         AT END MOVE HIGH-VALUES TO TRANS-SOC-SEC-NUMBER.

```

FIGURE 11.11 *Continued*

I2

**NONSEQUENTIAL
FILE
MAINTENANCE**

OVERVIEW

This chapter continues the discussion of file maintenance begun in Chapter 11. It differs significantly, however, in that the master file is accessed nonsequentially; i.e., one goes directly to the record being changed without having to read unaffected records in an old master file. In other words, a nonsequential update rewrites only those records which actually have activity, whereas a sequential update copies every record from an old master to a new file, even if it didn't change.

The unit opens with a conceptual discussion of diskette organization and *indexed* files. It introduces additional COBOL elements necessary for this type of file organization, then presents three complete programs to create, print, and update an indexed file. Presentation of the logic in the latter application is facilitated through pseudocode and hierarchy charts.

DISKETTE ORGANIZATION

A diskette is divided into 77 concentric circles known as *tracks*. A track is further subdivided into 26 *sectors*. Each sector has the capacity to store 256 *bytes* (or characters) so that the capacity of a Model II diskette is:

$$\frac{256 \text{ bytes}}{\text{sector}} \times \frac{26 \text{ sectors}}{\text{track}} \times \frac{77 \text{ tracks}}{\text{diskette}} = 512,512 \frac{\text{bytes}}{\text{diskette}}$$

The relationship between tracks and sectors is shown in Figure 12.1.

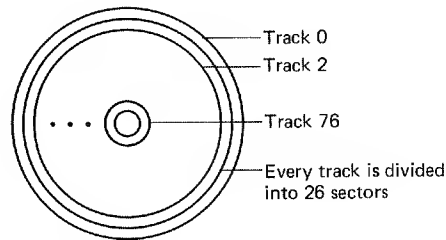


FIGURE 12.1 Organization of a diskette

The 77 sectors on a diskette are numbered from 0 to 76. In actuality, the capacity of track 0 is half that of the other 76 tracks, so that the true capacity of a Model II diskette is:

$$(256 \times 26 \times 76) + \frac{1}{2} (256 \times 26 \times 1) = 509,184$$

The exact numbers are not of particular importance. It is helpful, however, if the reader retains the track and sector concepts in order to better understand *indexed* file organization.

INDEXED FILES

The discussion that follows is an *intuitive* representation of how indexed files might be established under TRSDOS. It is included to provide insight as to how an indexed file permits both sequential and nonsequential access. An indexed file requires that records be loaded sequentially so that indexes can be established for direct access of individual records. Conceptually, there are two levels of indexing: a higher level or *track index*, and a lower level or *sector index*.

There is one track index for the entire file, and many sector indexes, one for each track in the file. *The track index contains the key of the highest*

<i>Track Number</i>	<i>Highest Key</i>
50	130
51	249
52	800
53	1240
54	1410
55	1811
56	2069
57	2410
58	2811
59	3040
60	3400
61	3619
62	4511
63	4900
64	5213
65	6874

FIGURE 12.2 Hypothetical track index

record contained in every track of the file. The sector index (for a given track) contains the key of the highest record for every sector in that track. As illustration, assume that an indexed file was loaded on tracks 50 through 65 of a diskette. A hypothetical track index (Figure 12.2) for this file contains 16 entries (one for each track).

Assume that record key 430 is to be retrieved. The record key, 430, is first compared to entries in the track index. Since an indexed file is loaded sequentially, record 430, *if it is present*, is in track 52. (The highest key in track 51 is 249, the highest key in track 52 is 800; hence, key 430 *if it is present* is in track 52.)

Next, the sector index for track 52 (Figure 12.3) is examined. *Each entry in the sector index contains the highest key for that sector.* (Observe that the last entry in the sector index has a key of 800, which matches the entry in the track index for track 52.) The highest keys on sectors 5 and 6 are 346 and 449, respectively; hence key 430, *if it is present*, is contained in sector 6 of track 52. The system now knows both the track and sector on which the record resides and may proceed directly to it.

Figure 12.3 implies that certain sectors within a track are reserved for special purposes. Sector 1, for example, stores the sector index itself. Sectors

<i>Sector</i>	<i>Highest Key</i>
1	Sector Index
2	268
3	289
4	301
5	346
6	449
7	491
8	516
.	.
20	800
21-26	Overflow

FIGURE 12.3 Hypothetical sector index for track 52

21-26 are overflow areas which accommodate records added to the file after it has been created.

When an indexed file is accessed, the I/O (input/output) routines of the operating system perform the preceding search for the programmer. The track and sector indexes are established automatically by the system when the file is loaded. The COBOL necessary to create and access indexed files is straightforward and should present little difficulty when introduced.

Let us address one last question before leaving this discussion: What happens when a new record is added to an existing indexed file? A major disadvantage of sequential organization is that the entire file has to be re-written if even a single record is added. One might logically ask, if indexed files are loaded sequentially, shouldn't a similar requirement pertain? Fortunately, the answer is no. The operating system anticipates the problem by preallocating space (overflow areas) in each track. When and if new records are added, linkages will be established so that the file is *logically* but *not necessarily physically in sequential order*. The details of this need not concern the reader. (Realize, however, that the normal TRSDOS PRINT utility will *not* work with an indexed file, because the file is not in strict sequential order. *Instead, one has to develop a separate COBOL program to print the contents of an indexed file.*)

We proceed to a discussion of the COBOL requirements for indexed files.

COBOL REQUIREMENTS FOR INDEXED FILES

Indexed files have additional COBOL requirements that center on the Environment and Procedure Divisions. The former uses an expanded SELECT statement. The latter uses new forms of the OPEN, READ, and WRITE, and introduces the REWRITE and DELETE verbs.

An abbreviated form of the SELECT statement associated with indexed files is:

```

SELECT file-name
  ASSIGN TO RANDOM external-file-name

  ORGANIZATION IS INDEXED

  [ ACCESS MODE IS { SEQUENTIAL
                    RANDOM } ]

  RECORD KEY IS data-name-1

  [ FILE STATUS IS data-name-2 ]

```

The ASSIGN clause specifies RANDOM (as opposed to INPUT or PRINT which have been used throughout). It also references an external file on the diskette which contains the data itself.

ORGANIZATION IS INDEXED is required to inform the compiler that an indexed file will be used, and that complex I/O routines will be required. Everything else having to do with adding, deleting, or changing records is handled automatically by routines in the operating system.

ACCESS MODE can be either SEQUENTIAL or RANDOM (that is, an indexed file can still have its records processed sequentially). Omission of this clause defaults to sequential access, and records will be read in ascending order of the record key. Realize, however, that when an indexed file is initially created, its records must be in sequential order.

The RECORD KEY clause indicates the field in each record on which the file is ordered and is used to build indexes for the operating system. Each record in the file requires a *unique* key.

FILE STATUS is used in conjunction with DECLARATIVES and is explained in that section.

A new option of the OPEN verb, OPEN I-O, is sometimes required. Consider first the syntax of the OPEN statement:

$$\text{OPEN} \left\{ \begin{array}{c} \text{INPUT} \\ \text{OUTPUT} \\ \text{I-O} \end{array} \right\} \text{file-name}$$

INPUT and OUTPUT are used when an indexed file is accessed or created, respectively. However, when only a single file is used as both the old and new master, as for nonsequential maintenance, OPEN I-O is necessary. In other words, the single master file functions as both an input and an output file.

The READ verb has a new format:

READ file-name [INTO identifier]
[INVALID KEY imperative statement]

This option is used only for nonsequential access (as opposed to the AT END format for sequential access). A random READ is preceded by a MOVE statement, in which the desired value of RECORD KEY is moved to the data name designated as the RECORD KEY in the SELECT statement. *The INVALID KEY clause is activated if the specified key cannot be found in the indexed file.* For example:

```
MOVE 888888888 TO SOC-SEC-NUMBER.
READ INDEXED-FILE
  INVALID KEY DISPLAY "RECORD NOT FOUND".
```

The indexed file is randomly accessed for the record with social security number 888-88-8888 (assuming SOC-SEC-NUMBER was designated as the RECORD KEY). If that record does not exist, the INVALID KEY condition is raised.

The WRITE statement has an additional clause associated with it. Consider:

WRITE record-name [FROM identifier]
[INVALID KEY imperative statement]

Recall that when creating an indexed file incoming records are required to be in sequential order, and further that each record is to have a unique key. The INVALID KEY condition is raised if either of these requirements is violated. (The COBOL syntax indicates that this clause is optional, and the same effect can be achieved through use of the DECLARATIVES. The latter is discussed later in the chapter.)

The REWRITE verb is used to replace existing records in a file when that file has been opened as an I-O file. Its syntax is similar to that of the WRITE verb, as shown:

REWRITE record-name [FROM identifier]
[INVALID KEY imperative statement]

The INVALID KEY condition is indicated if the record key of the last record read does not match the key of the record to be replaced.

Finally, the DELETE statement removes a record from a file. Its syntax is simply:

```
DELETE file-name
[INVALID KEY imperative statement]
```

The DELETE statement can be used only on a file that was opened in the I-O mode. The complex I-O routines associated with indexed files have built in error processing routines, e.g., those triggered by the INVALID KEY clause. The programmer can supplement this action through use of DECLARATIVES.

DECLARATIVES DECLARATIVES consist of one or more special purpose *sections*, which appear at the beginning of the Procedure Division, before the first executable statement. The purpose of DECLARATIVES is to expand the normal error handling procedures of the operating system. It is especially useful in conjunction with indexed files. Consider Figure 12.4.

If an I/O error occurs during processing of INDEXED-FILE, control is transferred to the appropriate DECLARATIVES section. Note well the USE statement in Figure 12.4, which indicates that the section will be executed only after an I/O error for INDEXED-FILE.

The INDEXED-ERROR section in turn contains executable code to display the value of two data names which were defined in Working-Storage and appeared in the SELECT statement for INDEXED-FILE. The FILE STATUS clause designated the two-byte area which is updated by the operating sys-

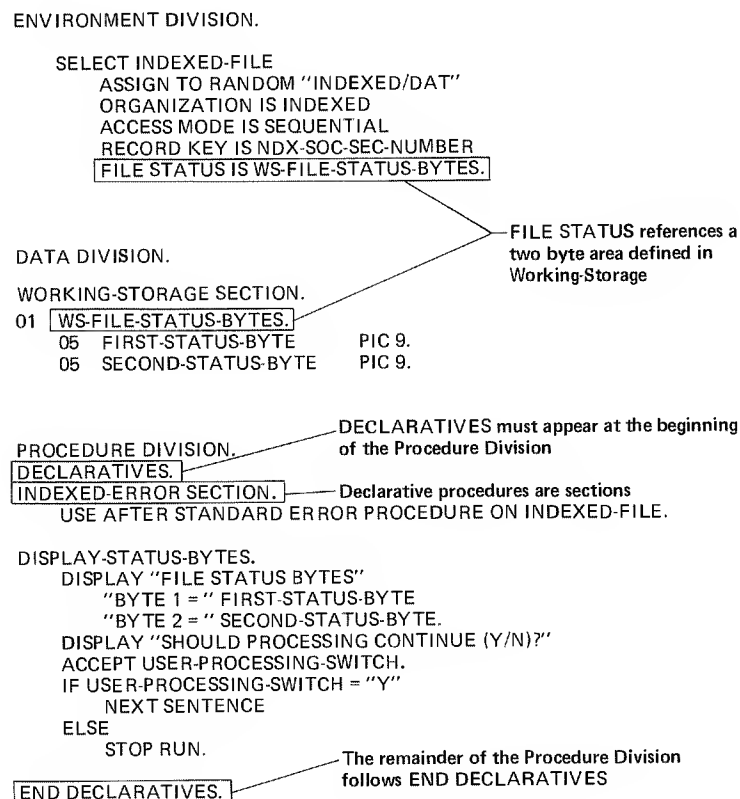


FIGURE 12.4 Use of DECLARATIVES

tem after *every* operation associated with the file. The user has the ability to interrogate these bytes to determine the success or failure of each operation. (The *COBOL Reference Manual*, catalog number 26-4703, explains the meaning of the various status codes.)

When an I/O problem occurs, control is transferred to DECLARATIVES, which displays the reason via the FILE STATUS bytes. The user is then given the opportunity to terminate or continue processing based on the value of these bytes. Consider now Figure 12.5 which creates an indexed file.

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID. INDEXDAT.
000120 AUTHOR. R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER. TRS-80.
000170 OBJECT-COMPUTER. TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210     SELECT INDEXED-FILE
000220         ASSIGN TO RANDOM "INDEXED/DAT"
000230         ORGANIZATION IS INDEXED
000240         ACCESS MODE IS SEQUENTIAL
000250         RECORD KEY IS NDX-SOC-SEC-NUMBER.
000260
000270 DATA DIVISION.
000280 FILE SECTION.
000290 FD INDEXED-FILE
000300     LABEL RECORDS ARE STANDARD
000310     RECORD CONTAINS 50 CHARACTERS
000320     DATA RECORD IS INDEXED-RECORD.
000330 01 INDEXED-RECORD.
000340     05 NDX-SOC-SEC-NUMBER PIC X(9).
000350     05 REST-OF-INDEXED-RECORD PIC X(41).
000360
000370 PROCEDURE DIVISION.
000380 MAINLINE.
000390     OPEN OUTPUT INDEXED-FILE.
000400     MOVE SPACES TO INDEXED-RECORD.
000410
000420     MOVE "11111111SMITH" ATL0105A180000781000000000"
000430     TO INDEXED-RECORD.
000440     WRITE INDEXED-RECORD
000450     INVALID KEY DISPLAY "ERROR ON ADDING INDEXED RECORD".
000460
000470     MOVE "22222222JONES" CHI0404E230000381200000980"
000480     TO INDEXED-RECORD.
000490     WRITE INDEXED-RECORD
000500     INVALID KEY DISPLAY "ERROR ON ADDING INDEXED RECORD".
000510
000520     MOVE "33333333BAKER" BOS0306P195000182180001280"
000530     TO INDEXED-RECORD.
000540     WRITE INDEXED-RECORD
000550     INVALID KEY DISPLAY "ERROR ON ADDING INDEXED RECORD".
000560
000570     MOVE "44444444MILGROM" DET0207G280000481240000480"
000580     TO INDEXED-RECORD.
000590     WRITE INDEXED-RECORD
000600     INVALID KEY DISPLAY "ERROR ON ADDING INDEXED RECORD".
000610
000620     MOVE "55555555CRAWFORD" WAS0305E300000581250001179"
000630     TO INDEXED-RECORD.
000640     WRITE INDEXED-RECORD
000650     INVALID KEY DISPLAY "ERROR ON ADDING INDEXED RECORD".
000660
000670     CLOSE INDEXED-FILE.
000680     STOP RUN.
000690

```

Denotes an indexed file

RECORD KEY clause references a data name defined within the indexed record

INDEXED-FILE is an output file

WRITE statement contains an INVALID KEY clause

RECORD KEYS are unique, and in ascending order

FIGURE 12.5 Program to create an indexed file

CREATING AN INDEXED FILE

Figure 12.5 contains a COBOL program to create an indexed file. It is similar in concept to Figure 3.5, which created a sequential data file.

The major difference between the two programs is the SELECT statement (lines 210-250 of Figure 12.5). Note well that the ASSIGN clause specifies a RANDOM file, although the file is accessed sequentially in this program. Observe also the inclusion of ORGANIZATION IS INDEXED. Of greatest significance is the RECORD KEY entry which refers to a field contained in the record description for INDEXED-FILE. The value of NDX-SOC-SEC-NUMBER must be *unique* for every record in the file; moreover, records must be in ascending order of NDX-SOC-SEC-NUMBER when the file is initially created. The RECORD KEY is used by the TRS-80 operating system to build the necessary indexes for the file. (The reader should observe the *unique* and *ascending* values in lines 420, 470, 520, 570, and 620.)

The WRITE statements of Figure 12.5 all contain an INVALID KEY clause. Recall that record keys for an indexed file must be unique and in ascending order when the file is created. The INVALID KEY clause will be triggered if either condition is not met.

PRINTING AN INDEXED FILE

The PRINT utility, introduced in Chapter 3, has been used exclusively to print the contents of a sequential file. The utility operates on data files (files which are input to, or output from, a program), print files (reports produced by a program), and even the output of the COBOL compiler, i.e., files with the extension of LST. The PRINT utility will *not*, however, work for an indexed file because the file is not strictly sequential.

The reader should convince himself that PRINT will *not* work with an indexed file. This is readily accomplished by compiling and subsequently running the program of Figure 12.5 to create the indexed file, INDEXED/DAT. After INDEXED/DAT has been created, attempt the command PRINT INDEXED/DAT and observe what happens.

The contents of an indexed file are printed only by writing and executing a COBOL program. An example of such a program is shown in Figure 12.6. The logic of Figure 12.6 is trivial, and virtually nothing is new about the program. The SELECT statement is the same as in Figure 12.5. Observe the ACCESS MODE IS SEQUENTIAL; i.e., we are reading the indexed file from start to finish in order of the record key.

The Procedure Division of Figure 12.6 opens the indexed file as an input file. (It was an output file in the previous program when it was created.) Lines 440 and 450 contain an initial READ statement, after which WRITE-INDEXED-RECORD is performed until the file is exhausted. Note that the last statement of the performed routine is another READ statement, which corresponds to the structure followed throughout the book.

The reader should attempt to compile and run the program of Figure 12.6, and obtain output corresponding to Figure 12.7. Observe the message COBOL STOP RUN AT: 0020 IN INDEXPRT. The 0020 is a reference to a hexadecimal address 0020 which corresponds to COBOL line 490 in Figure 12.6. Note also the five records that appear in Figure 12.7 correspond to the data embedded in Figure 12.5.

LOGICAL REQUIREMENTS OF THE MAINTENANCE PROGRAM

Given a basic understanding of indexed files and how these capabilities are implemented in COBOL, we return to the maintenance requirements of the update program.

The desired result of the nonsequential maintenance program to be developed in this chapter is virtually identical to that of the sequential main-

```

LINE  DEBUG PG/LN  A...B.....ID.....
1      000100 IDENTIFICATION DIVISION.
2      000110 PROGRAM-ID.  INDEXPRT.
3      000120 AUTHOR.      R GRAUER.
4      000130
5      000140 ENVIRONMENT DIVISION.
6      000150 CONFIGURATION SECTION.
7      000160 SOURCE-COMPUTER.    TRS-80.
8      000170 OBJECT-COMPUTER.   TRS-80.
9      000180
10     000190 INPUT-OUTPUT SECTION.
11     000200 FILE-CONTROL.
12     000210
13     000220 SELECT INDEXED-FILE
14     000230     ASSIGN TO RANDOM "INDEXED/DAT"
15     000240     ORGANIZATION IS INDEXED
16     000250     ACCESS MODE IS SEQUENTIAL
17     000260     RECORD KEY IS NDX-SOC-SEC-NUMBER.
18     000270 DATA DIVISION.
19     000280 FILE SECTION.
20     000290 FD  INDEXED-FILE
21     000300     LABEL RECORDS ARE STANDARD
22     000310     RECORD CONTAINS 50 CHARACTERS
23     000320     DATA RECORD IS INDEXED-RECORD.
24     000330 01  INDEXED-RECORD.
25     000340     05 NDX-SOC-SEC-NUMBER          PIC X(9).
26     000350     05 REST-OF-INDEXXED-RECORD    PIC X(41).
27     000360
28     000370 WORKING-STORAGE SECTION.
29     000380 01  PROGRAM-SWITCHES.
30     000390     05 END-OF-INDEXXED-FILE-SWITCH  PIC X(3)  VALUE SPACES.
31     000400
32     000410 PROCEDURE DIVISION.
33 >0000 000420 MAINLINE.
34 >0000 000430 OPEN INPUT INDEXED-FILE.
35 >0006 000440 READ INDEXED-FILE
36     000450     AT END MOVE "YES" TO END-OF-INDEXXED-FILE-SWITCH.
37 >0010 000460 PERFORM WRITE-INDEXXED-RECORD
38     000470     UNTIL END-OF-INDEXXED-FILE-SWITCH = "YES".
39 >001A 000480 CLOSE INDEXED-FILE.
40 >0020 000490 STOP RUN.
41     000500
42 >0022 000510 WRITE-INDEXXED-RECORD.
43 >0022 000520 DISPLAY INDEXED-RECORD.
44 >0026 000530 READ INDEXED-FILE
45     000540     AT END MOVE "YES" TO END-OF-INDEXXED-FILE-SWITCH.
46     ZZZZZZ END PROGRAM.

```

SELECT statement for INDEXED-FILE

INDEXED-FILE is an input file

0020 is machine address where program stops

INDEXED-FILE is accessed sequentially

*** END OF FILE ***

FIGURE 12.6 Program to print an indexed file

tenance program from Chapter 11. The difference is in the access method and number of files. The sequential program used *two* distinct master files, an old and a new, in addition to the transaction file. Every record in the old master had to be rewritten to the new master regardless of whether it changed. The nonsequential program will use a *single* master file, representing both the old and the new master. Records that change are rewritten, whereas nothing will be done to records remaining the same.

The logic of the sequential update was driven by *comparing* the social security numbers of the current old master and transaction records, as depicted in the pseudocode of Figure 11.4. The nonsequential update will be driven solely by the transaction file. An incoming transaction is read and an attempt made to match the transaction social security number with that of an existing record in the master file. A match on a correction, deletion, or

TRSDOS READY

Command to execute the compiled program

RUNCOBOL INDEXPRT

TRS-80 Model II COBOL Runtime (RM/COBOL ver 1.3B)
Copyright 1980 by Tandy Corp. Licensed from Ryan-McFarland Corp.

11111111SMITH	ATL0105A180000781000000000	} Indexed file contains five records
22222222JONES	CHI0404E230000381200000980	
33333333BAKER	BOS0306P195000182180001280	
44444444MILGROM	DET0207G280000481240000480	
55555555CRAWFORD	WAS0305E300000581250001179	

COBOL STOP RUN AT: 0020 IN INDEXP

TRSDOS READY

Indicates the machine language address
of the last executed statement

FIGURE 12.7 Commands and output to execute print program

salary update indicates that an existing record will be rewritten. A match on an addition implies an error due to a duplicate add.

If the transaction social security number is *not* found in the master file, and the transaction code is an add, a new record will be added to the master file. No match on a correction, deletion, or salary update indicates the transaction social security number was miscopied. Processing will cease when the transaction file is empty. The pseudocode for a nonsequential update is depicted in Figure 12.8.

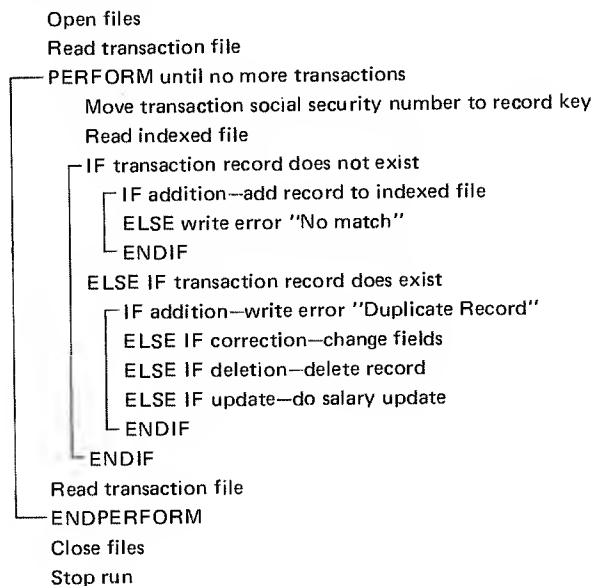


FIGURE 12.8 Pseudocode for nonsequential update

Figure 12.9 contains a hierarchy chart for the nonsequential update. As was stated earlier, the purpose of the hierarchy chart is to show the *function* of each routine in a program and the relationship among functions. It is not intended to show decision-making logic, which is the function of pseudocode.

The reader should observe that many of the module names found in Figure 12.9 also appeared in Figure 11.6, the hierarchy chart for sequential maintenance. PROCESS-MATCHING-IDS calls one of four lower level routines with identical functions to those of the sequential update. Note well, however, the presence of the module COPY-OLD-RECORD in Figure 11.5

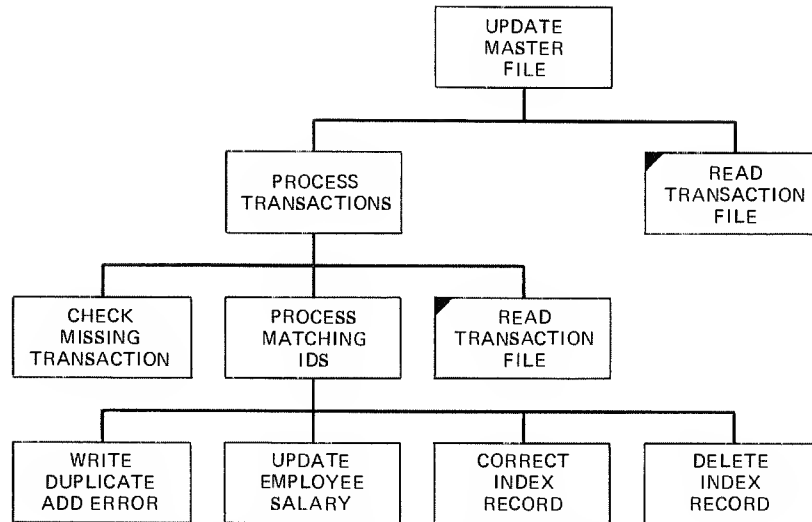


FIGURE 12.9 Hierarchy chart for nonsequential update

and its absence in Figure 12.9. This is because existing records in a sequential update must be copied to the new master whether or not they change. A nonsequential update, however, utilizes a single master file and rewrites only those records that actually change; it does not copy existing records with no activity, and hence there is no need for a COPY-OLD-RECORD routine. The COMPARE-IDS module is also missing from Figure 12.9, because a nonsequential update is driven by the transaction file rather than the relationship between the old master and transaction files.

UPDATING AN INDEXED FILE

The logic for nonsequential maintenance has been adequately detailed in the pseudocode of Figure 12.8, and the necessary COBOL has been discussed. Hence, we are ready to develop the program itself. Although the arguments for top down development and testing presented in Chapter 11 apply equally well in this chapter, we will not repeat that discussion. Accordingly, we proceed to the completed program.

Figure 12.10 illustrates the program to update an indexed file. The SELECT statement contains the FILE STATUS clause (line 290) which is used in conjunction with DECLARATIVES. The FILE STATUS entry designates a two-byte area, WS-FILE-STATUS-BYTES, which is defined in Working-Storage (line 840). The latter can be interrogated in DECLARATIVES, or anywhere else in the program, to monitor the result of I/O operations to INDEXED-FILE.

The indexed file is opened as I-O in line 980. This implies that the file will be read from, and written to, in the same program; i.e., it is an input/output file. Consequently, the nonsequential update contains one less file (three SELECT statements rather than four) than the sequential update of Figure 11.8.

The READ statement for the indexed file uses the INVALID KEY clause, lines 1150 and 1160. It is preceded by a MOVE statement in line 1130 which places the incoming social security number into the field designated as RECORD KEY in the SELECT statement. Note that the INVALID KEY condition, if raised, turns on a switch, hence the same switch is turned off immediately prior to the READ in line 1140.

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID.    INDEXUPD.
000120 AUTHOR.        R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER.    TRS-80.
000170 OBJECT-COMPUTER.    TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210     SELECT TRANSACTION-FILE
000220     ASSIGN TO INPUT "TRANS/DAT".
000230
000240     SELECT INDEXED-FILE
000250     ASSIGN TO RANDOM "INDEXED/DAT"
000260     ORGANIZATION IS INDEXED
000270     ACCESS MODE IS RANDOM
000280     RECORD KEY IS NDX-SOC-SEC-NUMBER
000290     FILE STATUS IS WS-FILE-STATUS-BYTES.
000300
000310     SELECT DELETED-RECORD-FILE
000320     ASSIGN TO OUTPUT "DELETED/DAT".
000330
000340 DATA DIVISION.
000350 FILE SECTION.
000360 FD  TRANSACTION-FILE
000370     LABEL RECORDS ARE STANDARD
000380     RECORD CONTAINS 51 CHARACTERS
000390     DATA RECORD IS TRANS-RECORD.
000400 01  TRANS-RECORD.
000410     05  TRANS-SOC-SEC-NUMBER          PIC X(9).
000420     05  TRANS-LAST-NAME              PIC X(15).
000430     05  TRANS-LOC-CODE               PIC X(3).
000440     05  TRANS-TITLE-CODE             PIC 999.
000450     05  TRANS-EDUCATION-CODE         PIC 9.
000460     05  TRANS-PERFORMANCE-CODE       PIC X.
000470     05  TRANS-SALARY                 PIC 9(5).
000480     05  TRANS-SALARY-MONTH           PIC 99.
000490     05  TRANS-SALARY-YEAR           PIC 99.
000500     05  FILLER                      PIC X(9).
000510     05  TRANS-CODE                  PIC X.
000520         88  ADDITION                  VALUE "A".
000530         88  CORRECTION                VALUE "C".
000540         88  DELETION                  VALUE "D".
000550         88  SALARY-UPDATE              VALUE "U".
000560
000570 FD  INDEXED-FILE
000580     LABEL RECORDS ARE STANDARD
000590     RECORD CONTAINS 50 CHARACTERS
000600     DATA RECORD IS INDEXED-RECORD.
000610 01  INDEXED-RECORD.
000620     05  NDX-SOC-SEC-NUMBER          PIC X(9).
000630     05  NDX-LAST-NAME              PIC X(15).
000640     05  NDX-LOC-CODE               PIC X(3).
000650     05  NDX-TITLE-CODE             PIC 999.
000660     05  NDX-EDUCATION-CODE         PIC 9.
000670     05  NDX-PERFORMANCE-CODE       PIC X.
000680     05  NDX-SALARY-DATA OCCURS 2 TIMES.
000690         10  NDX-SALARY              PIC 9(5).
000700         10  NDX-SALARY-MONTH        PIC 99.
000710         10  NDX-SALARY-YEAR        PIC 99.
000720
000730 FD  DELETED-RECORD-FILE
000740     LABEL RECORDS ARE STANDARD
000750     RECORD CONTAINS 50 CHARACTERS
000760     DATA RECORD IS DELETED-MASTER-RECORD.
000770 01  DELETED-MASTER-RECORD          PIC X(50).
000780

```

Same file as for sequential update

INDEXED-FILE will be accessed nonsequentially

References a two byte field defined in Working-Storage

FIGURE 12.10 Nonsequential update

```

000790 WORKING-STORAGE SECTION.
000800 01 PROGRAM-SWITCHES.
000810 05 TRANSACTIONS-REMAIN-SWITCH PIC X(3) VALUE SPACES.
000820 05 RECORD-NOT-FOUND-SWITCH PIC X(3) VALUE SPACES.
000830
000840 01 WS-FILE-STATUS-BYTES PIC XX.
000850
000860 PROCEDURE DIVISION.
000870 DECLARATIVES.
000880 I-O-ERROR SECTION.
000890 USE AFTER STANDARD ERROR PROCEDURE ON INDEXED-FILE.
000900 HANDLE-I-O-ERROR.
000910 DISPLAY "I/O ERROR OCCURRED".
000920 DISPLAY WS-FILE-STATUS-BYTES.
000930 END DECLARATIVES.
000940
000950 UPDATE-INDEXED-FILE SECTION.
000960 010-UPDATE-MASTER-FILE.
000970 OPEN INPUT TRANSACTION-FILE
000980 I-O INDEXED-FILE
000990 OUTPUT DELETED-RECORD-FILE.
001000
001010 PERFORM 190-READ-TRANSACTION-FILE.
001020
001030 PERFORM 020-PROCESS-TRANSACTIONS
001040 UNTIL TRANSACTIONS-REMAIN-SWITCH = "NO".
001050
001060 CLOSE TRANSACTION-FILE
001070 INDEXED-FILE
001080 DELETED-RECORD-FILE.
001090
001100 STOP RUN.
001110
001120 020-PROCESS-TRANSACTIONS.
001130 MOVE TRANS-SOC-SEC-NUMBER TO NDX-SOC-SEC-NUMBER.
001140 MOVE "NO" TO RECORD-NOT-FOUND-SWITCH.
001150 READ INDEXED-FILE
001160 INVALID KEY MOVE "YES" TO RECORD-NOT-FOUND-SWITCH.
001170
001180 IF RECORD-NOT-FOUND-SWITCH = "YES"
001190 PERFORM 100-CHECK-MISSING-TRANSACTION
001200 ELSE
001210 PERFORM 060-PROCESS-MATCHING-IDS.
001220
001230 PERFORM 190-READ-TRANSACTION-FILE.
001240 060-PROCESS-MATCHING-IDS.
001250 IF ADDITION
001260 PERFORM 070-WRITE-DUPLICATE-ADD-ERROR
001270 ELSE
001280 IF SALARY-UPDATE
001290 PERFORM 075-UPDATE-EMPLOYEE-SALARY
001300 ELSE
001310 IF CORRECTION
001320 PERFORM 080-CORRECT-NDX-RECORD
001330 ELSE
001340 IF DELETION
001350 PERFORM 090-DELETE-NDX-RECORD.
001360
001370 070-WRITE-DUPLICATE-ADD-ERROR.
001380 DISPLAY "ERROR - RECORD ALREADY EXISTS IN MASTER FILE: "
001390 TRANS-SOC-SEC-NUMBER.
001400
001410 075-UPDATE-EMPLOYEE-SALARY.
001420 IF NDX-SALARY (1) = TRANS-SALARY
001430 DISPLAY "ERROR - SALARY UPDATE ALREADY DONE: "
001440 TRANS-SOC-SEC-NUMBER
001450 ELSE
001460 MOVE NDX-SALARY-DATA (1) TO NDX-SALARY-DATA (2)
001470 MOVE TRANS-SALARY TO NDX-SALARY (1)

```

Defined in SELECT statement of INDEXED-FILE

DECLARATIVES appears at beginning of PROCEDURE DIVISION

Use of section name

INDEXED-FILE is both input and output

Nonsequential update is driven by the transaction file

Transaction key is moved to record key

INVALID KEY clause implies a random read

Last statement of performed routine reads the transaction file

FIGURE 12.10 Continued

```

001480      MOVE TRANS-SALARY-MONTH TO NDX-SALARY-MONTH (1)
001490      MOVE TRANS-SALARY-YEAR TO NDX-SALARY-YEAR (1)
001500      REWRITE INDEXED-RECORD.
001510
001520 080--CORRECT-NDX-RECORD.
001530      IF TRANS-LAST-NAME NOT = SPACES
001540          MOVE TRANS-LAST-NAME TO NDX-LAST-NAME.
001550      IF TRANS-LOC-CODE NOT = SPACES
001560          MOVE TRANS-LOC-CODE TO NDX-LOC-CODE.
001570      IF TRANS-TITLE-CODE NOT = SPACES
001580          MOVE TRANS-TITLE-CODE TO NDX-TITLE-CODE.
001590      IF TRANS-EDUCATION-CODE NOT = SPACES
001600          MOVE TRANS-EDUCATION-CODE TO NDX-EDUCATION-CODE.
001610      IF TRANS-PERFORMANCE-CODE NOT = SPACES
001620          MOVE TRANS-PERFORMANCE-CODE TO NDX-PERFORMANCE-CODE.
001630      IF TRANS-SALARY NOT = SPACES
001640          MOVE TRANS-SALARY TO NDX-SALARY (1).
001650      IF TRANS-SALARY-MONTH NOT = SPACES
001660          MOVE TRANS-SALARY-MONTH TO NDX-SALARY-MONTH (1).
001670      IF TRANS-SALARY-YEAR NOT = SPACES
001680          MOVE TRANS-SALARY-YEAR TO NDX-SALARY-YEAR (1).
001690
001700      REWRITE INDEXED-RECORD.
001710
001720 090--DELETE-NDX-RECORD.
001730      MOVE INDEXED-RECORD TO DELETED-MASTER-RECORD.
001740      WRITE DELETED-MASTER-RECORD.
001750      DELETE INDEXED-FILE.
001760
001770 100--CHECK-MISSING-TRANSACTION.
001780      IF ADDITION
001790          MOVE TRANS-SOC-SEC-NUMBER TO NDX-SOC-SEC-NUMBER
001800          MOVE TRANS-LAST-NAME TO NDX-LAST-NAME
001810          MOVE TRANS-LOC-CODE TO NDX-LOC-CODE
001820          MOVE TRANS-TITLE-CODE TO NDX-TITLE-CODE
001830          MOVE TRANS-EDUCATION-CODE TO NDX-EDUCATION-CODE
001840          MOVE TRANS-PERFORMANCE-CODE TO NDX-PERFORMANCE-CODE
001850          MOVE TRANS-SALARY TO NDX-SALARY (1)
001860          MOVE TRANS-SALARY-MONTH TO NDX-SALARY-MONTH (1)
001870          MOVE TRANS-SALARY-YEAR TO NDX-SALARY-YEAR (1)
001880          MOVE ZEROS TO NDX-SALARY-DATA (2)
001890          WRITE INDEXED-RECORD
001900      ELSE
001910          DISPLAY "ERROR - NO CORRESPONDING MASTER RECORD FOR "
001920              1 TRANS-SOC-SEC-NUMBER.
001930
001940 190--READ-TRANSACTION-FILE.
001950      READ TRANSACTION-FILE
001960      AT END MOVE "NO" TO TRANSACTIONS-REMAIN-SWITCH.

```

Use of REWRITE statement

Inactive records are physically removed from INDEXED-FILE

FIGURE 12.10 *Continued*

The program of Figure 12.10 is driven solely by the transaction file; i.e., one processes transactions until the transaction file is empty (lines 1030 and 1040). This is in contrast to the sequential update program of Figure 11.8 which was driven by the relationship of the old master and the transaction files.

The nonsequential update takes a record from the transaction file, and does a random read to determine if the transaction social security number exists on the indexed file (lines 1150-1160). If the record is not there, CHECK-MISSING-TRANSACTION will be performed; otherwise, if the transaction is present, PROCESS-MATCHING-IDS is executed.

The details of the lower level routines are similar to those of the sequential update. There are, however, two significant differences. First, records slated for deletion must be *explicitly* removed from the indexed

file through the DELETE verb (line 1750). This is in contrast to the sequential update which implicitly deletes records from the old master by not copying them to the new. Second, existing records which are changed are *rewritten* to the indexed file, lines 1500 and 1700. The sequential update, on the other hand, writes these records for the first time to the new master file.

Figure 12.10 also illustrates use of DECLARATIVES (lines 870-930). The word DECLARATIVES must appear immediately after the Procedure Division header. The end of DECLARATIVES is marked by the line END DECLARATIVES (line 930), which appears before the rest of the Procedure Division.

DECLARATIVES itself is divided into one or more sections; e.g., line 880. Line 890 contains a USE AFTER clause indicating that particular section is to be executed after the standard system error procedure for INDEXED-FILE. The particular example is rather trivial in that it merely displays the value of the FILE STATUS bytes; more complex examples could interrogate this field and take additional action based on specific values.

Finally, observe that the nondeclarative portion of the Procedure Division is also divided into sections, albeit a single section (line 950). This is because of a COBOL requirement which says that once sections are introduced in a program; e.g., in DECLARATIVES, they must be used throughout.

The program of Figure 12.10 was tested with the identical data as the sequential update of Chapter 11, and produced the same results as Figure 11.9. (The master files are physically different, but the conceptual output is identical.)

SUMMARY Indexed files were introduced in this chapter as a new means of file organization. The method is quite powerful and permits both sequential and nonsequential access.

The chapter began with a conceptual discussion of indexed files and the associated diskette organization. The necessary COBOL was introduced and three complete programs for creating, printing, and updating an indexed file were developed. The indexed file update had parallel requirements to the program from Chapter 11 and was documented with the aid of pseudocode and a hierarchy chart.

TRUE/FALSE

1. An active file is best updated sequentially.
2. An inactive file is best updated nonsequentially.
3. Sequential maintenance copies *every* record to the new master even if it doesn't change.
4. Nonsequential maintenance writes *only* those records which change to the new master.
5. Inactive indexed records are deleted through the REMOVE verb.
6. Incoming transactions must be in sequential order, even if processed nonsequentially.
7. Records in an indexed file need not have unique keys.
8. Records in an indexed file can be processed sequentially.

9. The RECORD KEY clause references a data name which is defined in Working-Storage.
10. New records are added to an indexed file with the ADD verb.

EXERCISES

Debugging: Figure 12.11 contains test data and invalid output produced by the program of Figure 12.12. The intended results should be the same as indicated by Figure 11.9. Find and correct all errors.

100000000BLACKBURN	CHI		C
222222222		250000981	U
333333333BAKER	DET	E 1281	C
400000000WICKS	CHI	0206A350000781	A
444444444MILGROM	DET	0207G280000481240000480A	A
555555555CRAWFORD			D
777777777SAMUEL	ATL	0407E300000881	A

Salary update was not done

Performance correction was not done

a) Transaction File

111111111SMITH	ATL0105A180000781000000000
222222222JONES	CHI0404E230000381200000980
333333333BAKER	BOS0306P195000182180001280
444444444MILGROM	DET0207G280000481240000480
555555555CRAWFORD	WAS0305E300000581250001179

b) Old Master (Indexed File)

111111111SMITH	ATL0105A180000781000000000
222222222JONES	CHI0404E230000381200000980
333333333BAKER	DET0306P1950001281180001280
400000000WICKS	CHI0206A350000781000000000
444444444MILGROM	DET0207G280000481240000480
555555555CRAWFORD	WAS0305E300000581250001179
777777777SAMUEL	ATL0407E300000881000000000

Record should have been deleted

c) New Master (Indexed File)

555555555CRAWFORD	WAS0305E300000581250001179
-------------------	----------------------------

d) Deleted File

```

ERROR - NO CORRESPONDING MASTER RECORD FOR 100000000
ERROR - RECORD ALREADY EXISTS IN MASTER FILE: 444444444
ERROR - RECORD ALREADY EXISTS IN MASTER FILE: 777777777

```

Duplicate add message should not appear

e) Error Message

FIGURE 12.11 Incorrect output of nonsequential update. (a) Transaction file. (b) Old master (indexed file). (c) New master (indexed file). (d) Deleted file. (e) Error message.

```

000100 IDENTIFICATION DIVISION.
000110 PROGRAM-ID. EINDEXUP.
000120 AUTHOR. R GRAUER.
000130
000140 ENVIRONMENT DIVISION.
000150 CONFIGURATION SECTION.
000160 SOURCE-COMPUTER. TRS-80.
000170 OBJECT-COMPUTER. TRS-80.
000180
000190 INPUT-OUTPUT SECTION.
000200 FILE-CONTROL.
000210     SELECT TRANSACTION-FILE
000220         ASSIGN TO INPUT "TRANS/DAT".
000230
000240     SELECT INDEXED-FILE
000250         ASSIGN TO RANDOM "INDEXED/DAT"
000260         ORGANIZATION IS INDEXED
000270         ACCESS MODE IS RANDOM
000280         RECORD KEY IS NDX-SOC-SEC-NUMBER
000290         FILE STATUS IS WS-FILE-STATUS-BYTES.
000300
000310     SELECT DELETED-RECORD-FILE
000320         ASSIGN TO OUTPUT "DELETED/DAT".
000330
000340 DATA DIVISION.
000350 FILE SECTION.
000360 FD TRANSACTION-FILE
000370     LABEL RECORDS ARE STANDARD
000380     RECORD CONTAINS 51 CHARACTERS
000390     DATA RECORD IS TRANS-RECORD.
000400 01 TRANS-RECORD.
000410     05 TRANS-SOC-SEC-NUMBER          PIC X(9).
000420     05 TRANS-LAST-NAME              PIC X(15).
000430     05 TRANS-LOC-CODE              PIC X(3).
000440     05 TRANS-TITLE-CODE            PIC 999.
000450     05 TRANS-EDUCATION-CODE        PIC 9.
000460     05 TRANS-PERFORMANCE-CODE      PIC X.
000470     05 TRANS-SALARY                PIC 9(5).
000480     05 TRANS-SALARY-MONTH          PIC 99.
000490     05 TRANS-SALARY-YEAR           PIC 99.
000500     05 FILLER                      PIC X(9).
000510     05 TRANS-CODE                  PIC X.
000520     88 ADDITION                    VALUE "A".
000530     88 CORRECTION                  VALUE "C".
000540     88 DELETION                    VALUE "D".
000550     88 SALARY-UPDATE               VALUE "X".
000560
000570 FD INDEXED-FILE
000580     LABEL RECORDS ARE STANDARD
000590     RECORD CONTAINS 50 CHARACTERS
000600     DATA RECORD IS INDEXED-RECORD.
000610 01 INDEXED-RECORD.
000620     05 NDX-SOC-SEC-NUMBER          PIC X(9).
000630     05 NDX-LAST-NAME              PIC X(15).
000640     05 NDX-LOC-CODE              PIC X(3).
000650     05 NDX-TITLE-CODE            PIC 999.
000660     05 NDX-EDUCATION-CODE        PIC 9.
000670     05 NDX-PERFORMANCE-CODE      PIC X.
000680     05 NDX-SALARY-DATA OCCURS 2 TIMES.
000690         10 NDX-SALARY              PIC 9(5).
000700         10 NDX-SALARY-MONTH        PIC 99.
000710         10 NDX-SALARY-YEAR        PIC 99.
000720
000730 FD DELETED-RECORD-FILE
000740     LABEL RECORDS ARE STANDARD
000750     RECORD CONTAINS 50 CHARACTERS
000760     DATA RECORD IS DELETED-MASTER-RECORD.
000770 01 DELETED-MASTER-RECORD          PIC X(50).
000780
000790 WORKING-STORAGE SECTION.

```

FIGURE 12.12 Invalid nonsequential update program

```

000800 01 PROGRAM-SWITCHES.
000810 05 TRANSACTIONS-REMAIN-SWITCH PIC X(3) VALUE SPACES.
000820 05 RECORD-NOT-FOUND-SWITCH PIC X(3) VALUE SPACES.
000830
000840 01 WS-FILE-STATUS-BYTES PIC XX.
000850
000860 PROCEDURE DIVISION.
000870 DECLARATIVES.
000880 I-O-ERROR SECTION.
000890 USE AFTER STANDARD ERROR PROCEDURE ON INDEXED-FILE.
000900 HANDLE-I-O-ERROR.
000910 DISPLAY "I/O ERROR OCCURRED".
000920 DISPLAY WS-FILE-STATUS-BYTES.
000930 END DECLARATIVES.
000940
000950 UPDATE-INDEXED-FILE SECTION.
000960 010-UPDATE-MASTER-FILE.
000970 OPEN INPUT TRANSACTION-FILE
000980 I-O INDEXED-FILE
000990 OUTPUT DELETED-RECORD-FILE.
001000
001030 PERFORM 020-PROCESS-TRANSACTIONS
001040 UNTIL TRANSACTIONS-REMAIN-SWITCH = "NO".
001050
001060 CLOSE TRANSACTION-FILE
001070 INDEXED-FILE
001080 DELETED-RECORD-FILE.
001090
001100 STOP RUN.
001110
001120 020-PROCESS-TRANSACTIONS.
001125 PERFORM 190-READ-TRANSACTION-FILE.
001130 MOVE TRANS-SOC-SEC-NUMBER TO NDX-SOC-SEC-NUMBER.
001140 MOVE "NO" TO RECORD-NOT-FOUND-SWITCH.
001150 READ INDEXED-FILE
001160 INVALID KEY MOVE "YES" TO RECORD-NOT-FOUND-SWITCH.
001170 IF RECORD-NOT-FOUND-SWITCH = "YES"
001180 PERFORM 100-CHECK-MISSING-TRANSACTION
001190 ELSE
001200 PERFORM 060-PROCESS-MATCHING-IDS.
001230
001240 060-PROCESS-MATCHING-IDS.
001250 IF ADDITION
001260 PERFORM 070-WRITE-DUPLICATE-ADD-ERROR
001270 ELSE
001280 IF SALARY-UPDATE
001290 PERFORM 075-UPDATE-EMPLOYEE-SALARY
001300 ELSE
001310 IF CORRECTION
001320 PERFORM 080-CORRECT-NDX-RECORD
001330 ELSE
001340 IF DELETION
001350 PERFORM 090-DELETE-NDX-RECORD.
001360
001370 070-WRITE-DUPLICATE-ADD-ERROR.
001380 DISPLAY "ERROR - RECORD ALREADY EXISTS IN MASTER FILE: "
001390 TRANS-SOC-SEC-NUMBER.
001400
001410 075-UPDATE-EMPLOYEE-SALARY.
001420 IF NDX-SALARY (1) = TRANS-SALARY
001430 DISPLAY "ERROR - SALARY UPDATE ALREADY DONE: "
001440 TRANS-SOC-SEC-NUMBER
001450 ELSE
001460 MOVE NDX-SALARY-DATA (1) TO NDX-SALARY-DATA (2)
001470 MOVE TRANS-SALARY TO NDX-SALARY (1)
001480 MOVE TRANS-SALARY-MONTH TO NDX-SALARY-MONTH (1)
001490 MOVE TRANS-SALARY-YEAR TO NDX-SALARY-YEAR (1)
001500 REWRITE INDEXED-RECORD.
001510
001520 080-CORRECT-NDX-RECORD.
001530 IF TRANS-LAST-NAME NOT = SPACES

```

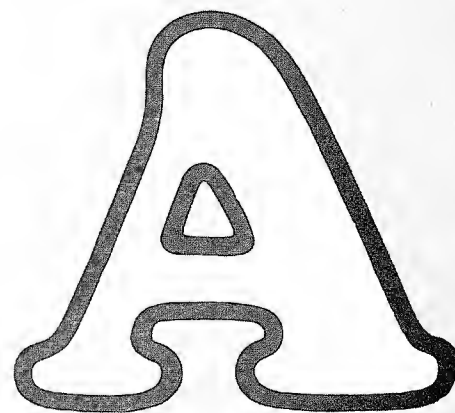
FIGURE 12.12 *Continued*

```

001540         MOVE TRANS-LAST-NAME TO NDX-LAST-NAME.
001550     IF TRANS-LOC-CODE NOT = SPACES
001560         MOVE TRANS-LOC-CODE TO NDX-LOC-CODE.
001570     IF TRANS-TITLE-CODE NOT = SPACES
001580         MOVE TRANS-TITLE-CODE TO NDX-TITLE-CODE.
001590     IF TRANS-EDUCATION-CODE NOT = SPACES
001600         MOVE TRANS-EDUCATION-CODE TO NDX-EDUCATION-CODE
001610     IF TRANS-PERFORMANCE-CODE NOT = SPACES
001620         MOVE TRANS-PERFORMANCE-CODE TO NDX-PERFORMANCE-CODE.
001630     IF TRANS-SALARY NOT = SPACES
001640         MOVE TRANS-SALARY TO NDX-SALARY (1).
001650     IF TRANS-SALARY-MONTH NOT = SPACES
001660         MOVE TRANS-SALARY-MONTH TO NDX-SALARY-MONTH (1).
001670     IF TRANS-SALARY-YEAR NOT = SPACES
001680         MOVE TRANS-SALARY-YEAR TO NDX-SALARY-YEAR (1).
001690
001700     REWRITE INDEXED-RECORD.
001710
001720 090-DELETE-NDX-RECORD.
001730     MOVE INDEXED-RECORD TO DELETED-MASTER-RECORD.
001740     WRITE DELETED-MASTER-RECORD.
001760
001770 100-CHECK-MISSING-TRANSACTION.
001780     IF ADDITION
001790         MOVE TRANS-SOC-SEC-NUMBER TO NDX-SOC-SEC-NUMBER
001800         MOVE TRANS-LAST-NAME TO NDX-LAST-NAME
001810         MOVE TRANS-LOC-CODE TO NDX-LOC-CODE
001820         MOVE TRANS-TITLE-CODE TO NDX-TITLE-CODE
001830         MOVE TRANS-EDUCATION-CODE TO NDX-EDUCATION-CODE
001840         MOVE TRANS-PERFORMANCE-CODE TO NDX-PERFORMANCE-CODE
001850         MOVE TRANS-SALARY TO NDX-SALARY (1)
001860         MOVE TRANS-SALARY-MONTH TO NDX-SALARY-MONTH (1)
001870         MOVE TRANS-SALARY-YEAR TO NDX-SALARY-YEAR (1)
001880         MOVE ZEROS TO NDX-SALARY-DATA (2)
001890         WRITE INDEXED-RECORD
001900     ELSE
001910         DISPLAY "ERROR -- NO CORRESPONDING MASTER RECORD FOR "
001920             TRANS-SOC-SEC-NUMBER.
001930
001940 190-READ-TRANSACTION-FILE.
001950     READ TRANSACTION-FILE
001960         AT END MOVE "NO" TO TRANSACTIONS-REMAIN-SWITCH.

```

FIGURE 12.12 *Continued*



**ANSWERS TO
TRUE/FALSE
AND EXERCISES**

CHAPTER 1

True/False

1. True.
2. False. They must be in sequence; Identification, Environment, Data, and Procedure.
3. False. They may contain any character, including reserved words, data names, paragraph names, etc.
4. False. Numeric literals may contain only numbers, a sign, or implied decimal point.
5. True.
6. True.
7. True.
8. False. Data names may, and frequently do, contain hyphens.
9. False. It terminates the IF and is of critical importance.
10. False. Reserved words are restricted to a specific context, and may not be used elsewhere.

Exercises

1. (a) IDENTIFICATION, AUTHOR, CONFIGURATION, SECTION, DIVISION, WORKING-STORAGE, etc. (See Appendix B for a complete list.)
(b) TEST-1, TEST-2, TEST-3, TOTAL-SCORE, AVERAGE, THE-BOSS, GET-INPUT, DO-PROCESSING, WRITE-OUTPUT
(c) "ENTER GRADE ON TEST 1"
"YOUR AVERAGE GRADE ON 3 TESTS = "
"YOU SHOULD STUDY HARDER"
(d) 3 (in line 490), 89 (in line 540), and 70 (in line 570).
(e) > and < in lines 540 and 570, respectively.
(f) 77
(g) PIC 999
(h) ACCEPT, DISPLAY, ADD, DIVIDE, PERFORM, IF, and STOP RUN.
2. (a) Change the numeric literal to 60 in line 570.
(b) Delete lines 540 and 550.
(c) Add a DISPLAY and ACCEPT statement for TEST-4 after line 450. Include TEST-4 in the ADD statement of line 480 and change 3 to 4 in the DIVIDE statement of line 490. Finally, do not forget to define the data name TEST-4 as a 77 level entry in Working-Storage.
3. (a) 567 Valid numeric literal.
(b) 567. Invalid: unless decimal point is intended as period at end of sentence.
(c) -567 Valid numeric literal.
(d) +567 Valid numeric literal.
(e) FIVE-SIX-SEVEN Invalid: non-numeric literals require quotes.
(f) "567." Valid non-numeric literal.
(g) "FIVE SIX SEVEN" Valid non-numeric literal.

- | | |
|-------------|--|
| (h) "-567." | Valid non-numeric literal. |
| (i) 567- | Invalid: a numeric literal cannot end with a sign. |
| (j) 567+ | Invalid: a numeric literal cannot end with a sign. |
| (k) "567+" | Valid non-numeric literal. |

4. The following are *invalid* data names:

- (b) DATE (Reserved word.)
- (c) 12345 (Does not begin with a letter.)
- (d) ONE TWO THREE (Spaces are not permitted.)
- (f) IDENTIFICATION (Reserved word.)
- (g) HOURS- (May not end on hyphen.)
- (i) GROSS-PAY-IN-\$ (\$ not allowed.)

Note: IDENTIFICATION-DIVISION in part (e) *is* permitted as a data name although it consists of two reserved words, IDENTIFICATION and DIVISION.

CHAPTER 2

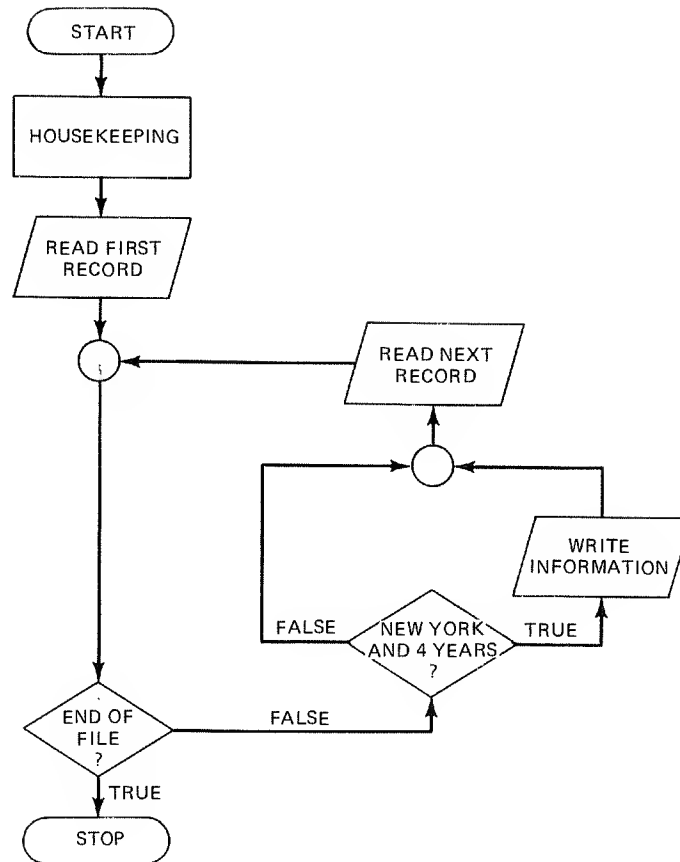
True/False

- 1. True.
- 2. False. A file is a set of records.
- 3. False. The computer does exactly what it is told to do.
- 4. False. A record contains one or more fields.
- 5. False. Indentation, capitalization, and so on are at programmer discretion.
- 6. True.
- 7. True.
- 8. True.
- 9. False. A file name typically appears in a SELECT, FD, OPEN, CLOSE, and READ statement.
- 10. False. It is required only when files are processed, which is usually the case.

Exercises

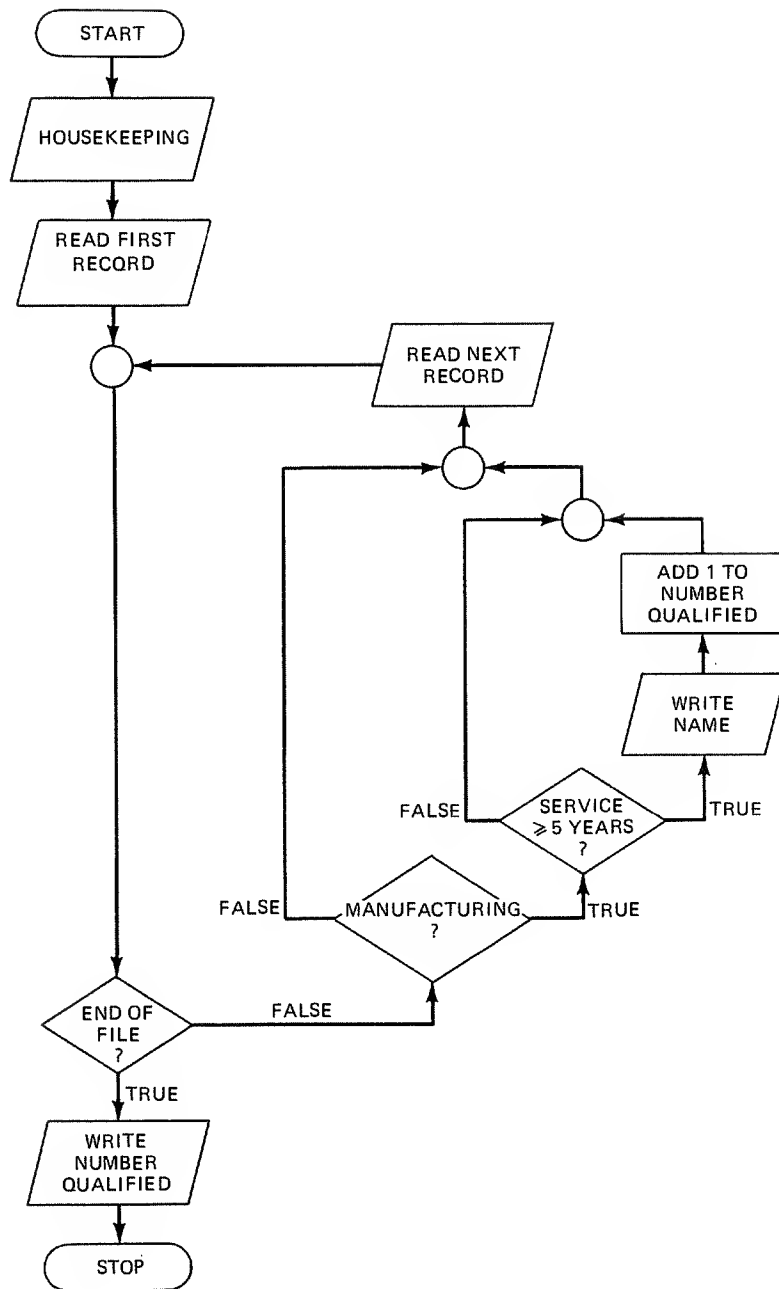
- 1. (a) New York is spelled several different ways in the incoming data. This can cause difficulty in selecting qualified records in the same fashion as the two spellings of programmer. Secondly, date of hire rather than years of service should be stored with incoming records. The latter field can be calculated from the former.
- (b) Housekeeping
 - Initial read
 - Do while data remains
 - If service is 4 years or more and location is New York
 - Write name on report
 - Else
 - Do nothing
 - End Do
 - Stop

(b) continued



- (c) Assuming that all spellings of New York are acceptable, Barbarino, Horshack, and Albright qualify. Welby is in New York but fails the service test. Anderson, Kotter, Unger, and Madison fail the location test.
- (d) No — assuming that the input table represents all available data because information was not collected on age or birth date.

2.



3. (1) CONFIGURATION SECTION.
 (2) STUDENT-FILE
 (3) FILE SECTION.
 (4) 80
 (5) FD
 (6) 01
 (7) 77
 (8) INPUT
 (9) CLOSE
 (10) PROCESS-RECORDS.
 (11) STUDENT-FILE
 (12) WS-DATA-REMAINS-SWITCH.

CHAPTER 3

True/False

1. False. RSCOBOL invokes the COBOL compiler. RUNCOBOL executes a compiled program.
2. True.
3. False. One must specify the compiler option X.
4. False. X is a conditional change and requires a confirming user response.
5. False. It changes the first 10 occurrences of PRT irrespective of line numbers.
6. True.
7. False. It prints lines 100 through 300 which may be more or less than 20 lines.
8. False. The N command is used to renumber a program; e.g., N 100,10.
9. True.
10. False. The utility program is PRINT.

Exercises

1. (a) A margin
(b) * in column 7
(c) A margin
(d) B margin
(e) A margin
(f) A margin
(g) A margin
(h) B margin
(i) B margin
(j) B margin
(k) B margin
(l) A margin
(m) B margin

Note: The A margin is columns 8 to 11; the B margin begins in column 12.

2. g-1
d-2
a-3
b-4
e-5
c-6
f-7
i-8
h-9
j-10

3.

	<i>Memory Contents</i>							
	<i>Before</i>				<i>After</i>			
	500	600	700	ACC	500	600	700	ACC
LOAD 500	10	20	?	?	10	20	?	10
MULTIPLY 600	10	20	?	10	10	20	?	200
STORE 700	10	20	?	200	10	20	200	200

4. The COBOL instruction: ADD A B C GIVING D, takes the value of A, adds it to B, adds this sum to C, and puts the result in D. Assume A, B, C, and D are stored in locations 100, 200, 300, and 400 respectively. The following sequence of machine instructions could be generated:

<i>Instruction</i>	<i>Comments</i>
LOAD 100	Brings A into accumulator.
ADD 200	Adds B to A.
ADD 300	Adds C to sum of A and B.
STORE 400	Stores A+B+C in D.

5. First enter the editor with the command CEDIT, after seeing the TRSDOS READY message. Next, load the file in question, L FIRSTDAT. Delete lines 440 and 450 to eliminate Marshal Crawford. Enter the insert mode, e.g., I 532,1 to establish a MOVE and corresponding WRITE for David Brown. Change Marion Milgrom's age by first positioning the appropriate line, e.g., P 520, then X/24/33/. Finally, don't forget to write the changed file, i.e., W FIRSTDAT.
6. The necessary commands:

```

CEDIT
L PREAMBLE
C/UNITEDS/UNITED S/2
B
C/ONION/UNION/
D 121
E 130 (Delete 7 characters; DDDDDDD, then return)
I 160
OF AMERICA
B
P

```

CHAPTER 4

True/False

1. True.
2. False. There are two distinct ADD statement formats, one with GIVING, the other with TO.
3. True.
4. True.
5. True.
6. False. It can be written with or without an ELSE.
7. False. All statements, up to a period or ELSE, are executed.
8. False. It is the last statement executed, but can appear anywhere in the Procedure Division.
9. False. A file name appears in SELECT, FD, OPEN, CLOSE, and READ statements.
10. True.

Exercises

1. The completed table is shown below. Hyphens indicate that a value remained unchanged as the result of the instruction.

Data-name	A	B	C	D
Value before execution:	4	8	12	1
Value after execution of:				
ADD 1 TO D.	—	—	—	2
ADD A B C GIVING D.	—	—	—	24
ADD A B C TO D.	—	—	—	25
SUBTRACT A B FROM C.	—	—	0	—
SUBTRACT A B FROM C GIVING D.	—	—	—	0
MULTIPLY A BY B.	—	32	—	—
MULTIPLY B BY A.	32	—	—	—
DIVIDE A INTO C.	—	—	3	—
DIVIDE C BY A GIVING B.	—	3	—	—
DIVIDE C BY B GIVING D.	—	—	—	1.5
COMPUTE D = A + B / 2 * D.	—	—	—	8
COMPUTE D = (A + B) / (2 * D).	—	—	—	6
COMPUTE D = A + B / (2 * D).	—	—	—	8
COMPUTE D = (A + B) / 2 * D.	—	—	—	6
COMPUTE D = A + (B / 2) * D.	—	—	—	8

2.	Sending Field		Receiving Field	
	Picture	Contents	Picture	Contents
(a)	9(6)	123456	9(6)	123456
(b)	9(6)	123456	9(8)	00123456
(c)	9(6)	123456	9(6).99	123456.00
(d)	9(4)V99	123456	9(6)	001234
(e)	9(4)V99	123456	9(4)	1234
(f)	9(4)V99	123456	\$\$\$\$9.99	—\$1234.56
(g)	9(4)V99	123456	\$\$\$,\$9.99	—\$1,234.56
(h)	9(6)	123456	\$\$\$\$,\$9.99	\$123,456.00

3. i. Group items are all those *without* a Picture clause; i.e., EMPLOYEE-RECORD, EMPLOYEE-NAME, BIRTH-DATE, EMPLOYEE-ADDRESS, NUMBER-AND-STREET, and CITY-STATE-ZIP.
- ii. All elementary items have a Picture clause. These are: SOC-SEC-NUMBER, LAST-NAME, FIRST-NAME, MIDDLE-INIT, BIRTH-MONTH, BIRTH-DAY, BIRTH-YEAR, HOUSE-NUMBER, STREET-NAME, CITY, STATE, and ZIP.
- FILLER would also be considered an elementary item.
- iii. (a) Columns 1–9 (j) Columns 43–77
 (b) Columns 10–32 (k) Columns 43–58
 (c) Columns 10–21 (l) Columns 43–48
 (d) Columns 22–31 (m) Columns 49–58
 (e) Column 32 (n) Columns 59–77
 (f) Columns 34–39 (o) Columns 59–68
 (g) Columns 34–35 (p) Columns 69–72
 (h) Columns 36–37 (q) Columns 73–77
 (i) Columns 38–39
4. (a) Invalid: one opens a file, not a record.
 (d) Invalid: the type of file is not specified in a CLOSE statement.
 (e) Invalid: the READ statement should have an AT END clause.
 (g) Invalid: one reads a file, rather than a record.
 (i) Invalid: 2 should replace TWO.

- (l) Invalid: one writes a record, rather than a file.
 - (m) Invalid: the AT END clause is not permitted in a WRITE statement.
 - (p) Invalid: either AFTER or BEFORE are required.
5. (a) COMPUTE X = A + B + C.
 (b) COMPUTE X = (A + B * C) / 2.
 (c) COMPUTE X = ((A * B) + (C * D)) / (E * F).
 (d) COMPUTE X = (A + B) / 2 - C.

CHAPTER 5

True/False

1. False. Blanks are the delimiters between COBOL elements and are not permitted.
2. False. Correct compilation means that the program has been translated into machine language.
3. True.
4. False. Compilation will resume with a warning message.
5. False. Most do, but some appear at the end of the listing.
6. True.
7. False. Execution is possible, but will usually be wrong, and end prematurely.
8. False. The compiler detects only those errors which violate the COBOL syntax.
9. True.
10. True.

Exercises

1. Compilation errors:

<i>Line Number</i>	<i>Correction</i>
66	SYNTAX — Hyphens are required in TOTAL-UNION-FEE.
76	SYNTAX, DOUBLE DECLARATION, etc. — All problems stem from a missing period in line 75.
92	RESERVED WORD CONFLICT — START cannot be used as a paragraph name as it is a reserved word.
96	UNDEFINED — Should be STUDENT-FILE, rather than STUD-FILE.
112, 120, 122, 144	IDENTIFIER — CREDITS is not unique. This is best fixed by defining PRINT-CREDITS in line 40; also, the * in line 112 should be preceded by a space.
134	SYNTAX — The ADD statement cannot contain both GIVING and TO. The error is best corrected by deleting GIVING TOTAL-TUITION.
135, 158	IDENTIFIER — Will disappear with the previous fix to line 66.
137	DATA TYPE — TOTAL-IND-BILL should be defined with PIC 9(6) rather than X(6) in line 67.
150	REFERENCE INVALID — The WRITE statement requires a record name, PRINT-LINE, not a file name.

2. Execution errors:

- (a) The total amount of individual bills is incorrect in the total line: TOTAL-IND-BILL is defined in line 760 and correctly incremented for each record in line 1460. However, when the total line is built in lines 1650-1700, IND-BILL, rather than TOTAL-IND-BILL, is moved to PRINT-IND-BILL in line 1700.

- (b) The total for UNION-FEE is wrong: TOTAL-UNION-FEE is defined and initialized in line 750. However, when the other counters are incremented in lines 1430 to 1470, an ADD statement for TOTAL-UNION-FEE is missing. TOTAL-UNION-FEE is subsequently moved to PRINT-UNION-FEE in line 1670, but TOTAL-UNION-FEE never budged from its initial value of zero.
- (c) The last record was processed twice. Recall that when the program structure was first presented in Chapter 2, there was an *initial* READ statement in the mainline paragraph, and a *second* READ, as the *last* statement in the performed routine. That structure was correct. In Figure 5.6, the initial READ statement was eliminated, and the second READ incorrectly moved to the beginning of the performed routine.

To understand the effect, consider a file with only a single record. When the end of PROCESS-RECORDS is reached the first time, the end of file has not yet been sensed; hence PROCESS-RECORDS is entered a second time, even though there is only a single record. The end of file is sensed immediately in line 1201, but the perform is not terminated until line 1380. Consequently, the intermediate statements are executed a second time for the previous record. The problem is corrected by restoring an initial read in the mainline paragraph, between lines 1040 and 1070, and placing the existing READ statement of lines 1201 and 1202 after line 1380.

- (d) ACTIVITY-FEE computations are incorrect; consider the case of John Smith and his 15 credits. The value of IND-ACTIVITY-FEE is initially set to 25 in line 1280, to 75 in line 1320, and again reset to 50 in line 1322. The problem is simply that the IF statements are inverted, causing anyone with 6 credits or more to be charged \$50. Reversing these statements will set the activity fee to \$50 for students with 7 to 12 credits and to \$75 for anyone with more than 12 credits.
- (e) Individual SCHOLARSHIP is incorrect: in line 1570, SCHOLARSHIP with picture 9(4) is moved to PRINT-SCHOLARSHIP with picture \$\$9. The largest value that can appear in the latter field is \$99; hence, any scholarship amounts in excess of \$99 will have the high-order digits eliminated.

CHAPTER 6

True/False

1. True.
2. False. They are optional in COBOL, but may be necessary in a commercial environment.
3. False. Level numbers per se have no effect on the CORRESPONDING option.
4. False. Qualification may be required over several levels.
5. True.
6. True.
7. False. The ELSE clause is optional, and thus may not be present for every IF.
8. False. They are also known as 88-level entries.
9. True.
10. True.

Exercises

1. (a)

STATE - TABLE																											
STATE - NAME (1)										...	STATE - NAME (50)					POP (1)					...	POP (50)					
																</											

4. (a) THIRD-ROUTINE
(b) SECOND-ROUTINE
(c) FIRST-ROUTINE
(d) SECOND-ROUTINE
(e) FIRST-ROUTINE
(f) THIRD-ROUTINE
5. (a) True. FIELD-E does not appear in RECORD-ONE.
(b) False. FIELD-D has matching qualification, hence it will be moved.
(c) False. The CORRESPONDING option looks at data-names, rather than level numbers.
(d) False. The CORRESPONDING option looks at data-names, rather than positions.
(e) False. The two left-most bytes of FIELD-B will be moved.
6. First, DATE-WORK-AREA must be in the form YYMMDD:

```

01 DATE-WORK-AREA.
   05 TODAYS-YEAR    PIC 99.
   05 TODAYS-MONTH   PIC 99.
   05 TODAYS-DAY     PIC 99.

```

Second, the COMPUTE statement should be rewritten.

```

COMPUTE EMPLOYEE-AGE = TODAYS-YEAR - BIRTH-YEAR
+ (TODAYS-MONTH - BIRTH-MONTH) / 12.

```

The student should be urged to *plug numbers in* to both versions of the COMPUTE to be convinced of the difference.

CHAPTER 7

True/False

1. False. Sequencing should be required by the shop. It is not imposed by COBOL.
2. False. Blank lines enhance a program's appearance; e.g., before paragraph headers, 01 entries, etc.
3. True.
4. False. Paragraphs should be restricted to a single function; e.g., reading or computing.
5. True.
6. False. Comments may be redundant or inconsistent with source code; neither is desirable.
7. False. It is both possible and desirable to eliminate 77-level entries.
8. False. Such cryptic data names are impossible to understand by others and shouldn't be used.
9. False. Good indentation greatly enhances one's understanding of source code.
10. False. COBOL requires Picture clauses to begin in the B margin; the shop may require uniformity.

CHAPTER 8

True/False

1. False. They are equivalent, and print if and only if the sending field is negative.
2. False. They frequently appear together; the * is used for check protection.
3. False. Only a signed field (PIC S9 etc) can hold negative quantities.

4. False. Arithmetic can be performed only on fields with a 9, S, or V (i.e., numeric fields only).
5. True.
6. False. They must be presorted to correspond to the control break sequence.
7. False. They can be over several levels; the chapter in the book illustrated a two-level problem.
8. False. It is the responsibility of the programmer to initialize all counters.
9. False. They are called for frequently; hence, the entire chapter was devoted to the subject.
10. False. Both techniques are worth the effort so that complex programs may be followed more easily.

Exercises

1.	Source Field		Receiving Field	
	Picture	Value	Picture	Edited Result
(a)	S9(4)V99	-045600	\$\$\$\$.99CR	\$456.00CR
(b)	S9(4)V99	045600	\$\$\$\$.99DB	\$456.00
(c)	S9(4)	4567	\$\$\$\$.00	\$4,567.00
(d)	S9(6)	122577	99B99B99	12 25 77
(e)	S9(6)	123456	++++,+++	+123,456
(f)	S9(6)	-123456	++++,+++	-123,456
(g)	S9(6)	123456	----,---	123,456
(h)	S9(6)	-123456	----,---	-123,456
(i)	9(6)V99	00567890	\$\$\$\$\$.99	\$5,678.90
(j)	9(6)V99	00567890	\$\$\$Z,ZZZ.99	\$ 5,678.90
(k)	9(6)V99	00567890	\$***,***.99	\$**5,678.90

2. (a) AMOUNT-REMAINING is defined as an unsigned numeric field (PIC 9(3)), which by definition can never be less than zero.

(b) AMOUNT-REMAINING (Before Subtraction)	QUANTITY-SHIPPED	AMOUNT REMAINING (After Subtraction)
100	30	70
70	50	20
20	25	5
5	15	10

Note well that AMOUNT-REMAINING can never go negative; hence, when 25 is subtracted from 20, the answer is the absolute value of the subtraction; i.e., 5. In similar fashion, when 15 is subtracted from 5, the answer is 10.

3. *Debugging*: The first step in debugging any program is to identify the errors which have occurred. That has been done for the reader through the captions on Figure 8.10. We will address those errors in sequence.

The salesman total for Smith is missing the edit characters CR, to indicate that the total is negative. The first thought is that CR has been simply omitted from the edit picture (line 970 in Figure 8.11), but that proves not to be the case. The only conclusion possible is that the value of THIS-SALESMAN-TOTAL, the field which is moved to PRT-SALESMAN-TOTAL, is not negative. Remember, CR will print if and only if the sending field is negative.

THIS-SALESMAN-TOTAL is defined in line 470 as an *unsigned* numeric field; hence, it can never go negative and the problem is solved.

The difficulty with cumulative location total is simply that THIS-LOCATION-TOTAL is not reinitialized to zero for each new location. The only question is where to insert the statement, MOVE ZEROS TO THIS-LOCATION-TOTAL.

Examination of the hierarchy chart of Figure 8.7 suggests the module PROCESS-ALL-LOCATIONS. Closer scrutiny of the program in Figure 8.11 pinpoints line 1300.

The final difficulty involves printing the *previous* location and occurs in the module WRITE-LOCATION-HEADING. The data name TR-SALESMAN-LOCATION should be substituted for WS-PREVIOUS-LOCATION in line 1720.

CHAPTER 9

True/False

1. True.
2. False. They can, but need not be, identical.
3. False. It appears in the called (i.e., sub) program only.
4. False. It contains all four divisions of a regular COBOL program.
5. True.
6. True.
7. False. The COPY statement is frequently used to initialize a table.
8. True.
9. False. The order is critical!!!
10. True.

Exercises

1. The primary advantage of a COPY statement is that several programs have access to the *identical* table. Hence, if changes are necessary, they need only be made in one place; i.e., the copied module. The latter is changed the same way as an ordinary COBOL program using CEDIT. Realize, however, that any program which uses the copied module has to be *recompiled* after the changes have taken place.

2. *Debugging*: The problems of salesman and location total remaining at zero are related. The paragraph PROCESS-ALL-TRANSACTIONS, in Figure 9.8, is intended to increment (decrement) the appropriate salesman, location, and company totals. At first glance the paragraph appears correct. Closer examination, however, reveals asterisks in lines 1460, 1470, 1520, and 1530. Recall that an asterisk in column 7 indicates a comment causing the associated line to be ignored by the compiler. Hence, the aforementioned entries are *not* translated into executable code. The problem is easily solved by removing the asterisks.

The problem in the location heading leads to the subprogram of Figure 9.9 where the location code is expanded. The arguments are passed correctly between the main and subprograms via USING clauses in the CALL statement (line 1720 of Figure 9.8) of the main program and the Procedure Division header (line 490 of Figure 9.9) of the subprogram. The difficulty lies within the routine to expand the location code.

Lines 540-560 indicate that EXPAND-LOCATION-CODE will be executed repeatedly until WS-LOCATION-SWITCH equals "YES"; i.e., until a match is found or the table is exceeded. The first time the subroutine is called, a match is found for Atlanta, WS-LOCATION-SWITCH is set to YES, and the heading prints correctly. Unfortunately, WS-LOCATION-SWITCH is never reset so that EXPAND-LOCATION is not executed when the subprogram is reentered. The solution is to insert line 530, MOVE "NO" TO WS-LOCATION-SWITCH on Figure 9.9.

CHAPTER 10

True/False

1. True.
2. True.
3. True.
4. True.
5. False. It can't; hence, the need for the REDEFINES clause.
6. False. The table is better initialized by reading from a file which eliminates the REDEFINES.
7. False. One is the entry in row 4, column 1; the other in row 1, column 4.
8. False. It merely allocates space.
9. True.
10. True.

Exercises

1. (a) The 12 pairs of values are given below. Each set in parentheses represents (SUB1,SUB2):

$$\begin{array}{cccccc} (1,1), & (1,2), & (1,3), & (2,1), & (2,2), & (2,3) \\ (3,1) & (3,2), & (3,3), & (4,1), & (4,2), & (4,3) \end{array}$$

- (b) Replacing the greater-than signs by equal signs would result in performing 010-READ-CARDS a total of *six* times as follows:

(1,1) (1,2) (2,1) (2,2) (3,1) (3,2)

If less-than signs were used, the paragraph would not be performed at all, as the condition is satisfied immediately. Remember, **PERFORM/VARYING** increments, tests, and then branches.

2. (a)

ENROLLMENTS									
COLLEGE (1)						COLLEGE (2)			
SCHOOL (1)				SCHOOL (5)			SCHOOL (1)		
YR1		YR4		YR1		YR4	YR1		
			...						
			...						

A total of 320 ($4 \times 5 \times 4 \times 4$) locations are allocated. The first 80 are for College 1, the next 80 for College 2, etc.

(b)

ENROLLMENTS											
COLLEGE (1)						COLLEGE (2)					
SCH1		SCH5	YR1		YR4	SCH1					
...						

A total of 144 locations are allocated. The first 36 are for the first college, the next 36 for the second college, etc. Within each group of 36, the first 20 are for schools 1-5, and the last 16 for years 1-4.

3. (a) Valid.
 - (b) Valid.
 - (c) Invalid: REGION requires a single subscript.
 - (d) Valid syntactically; invalid logically as there are only 6 regions. (It will compile correctly, but present a problem during execution.)
 - (e) Invalid: CITY requires two subscripts.
 - (f) Valid.
 - (g) Valid.
 - (h) Valid syntactically; invalid logically.
4. The 24 sets of values are given below. Each set in parentheses represents (SUB1,SUB2,SUB3):

(1,1,1), (2,1,1), (3,1,1), (4,1,1)
 (1,2,1), (2,2,1), (3,2,1), (4,2,1)
 (1,1,2), (2,1,2), (3,1,2), (4,1,2)
 (1,2,2), (2,2,2), (3,2,2), (4,2,2)
 (1,1,3), (2,1,3), (3,1,3), (4,1,3)
 (1,2,3), (2,2,3), (3,2,3), (4,2,3)

5. There are three problems associated with the table lookups for location, title, and performance. We begin with location. Line 2070 of Figure 10.15 initializes a switch, lines 2080-2100 invoke the paragraph SEARCH-LOCATION-TABLE, and lines 2530-2600 do the actual table processing. All are *correct*, so the problem must lie elsewhere. Realize that *if nothing is wrong with the logic to do a table lookup, perhaps there is something wrong with the table itself*. Closer examination of lines 770-920 which establish the location table reveals that the REDEFINES clause is missing in line 890. Hence, LOCATION-VALUES and LOCATION-TABLE point to *different* areas in memory with the unfortunate result that the table was never properly initialized.

Analogous reasoning solves the title problem as well. The table lookup, lines 2130-2150 and 2620-2710, is *correct* in and of itself. The problem again lies in the table initialization. TITLE-TABLE is defined in lines 960-1010. It is to be initialized by reading values from a file, and the code in lines 1760-1940 appears correct. The error is subtle and stems from the fact that NUMBER-OF-TITLES is never incremented as each title is added to the table. (Note well the OCCURS DEPENDING ON clause in the definition of the table, line 980.) Line 2630 of the table lookup procedure includes a check to ensure that the table size is not exceeded. However, since NUMBER-OF-TITLES remains at zero, lines 2640-2650 are always executed. The solution — insert line 1900, ADD 1 TO NUMBER-OF-TITLES.

The problem of performance is traced directly to the nested IF of lines 2800-2890. An ELSE clause is missing in line 2850; hence, the IF of line 2860 is executed whenever performance is average. Obviously, a value of A for EMP-PERFORMANCE-CODE cannot simultaneously produce a value of P, causing all employees with an average rating to show as unknown.

The incorrect value of "months between increase" is traced to the COMPUTE statement of lines 2950-2970. A conversion from years to months is required; hence, line 2960 should be multiplied by 12.

CHAPTER 11

True/False

1. True.
2. False. Testing should begin as soon as possible with the aid of program stubs.
3. False. The higher level modules (with the more complex logic) are tested first.
4. False. They should be as independent as possible.
5. False. It has no syntax per se; indentation, capitalization, etc., are at programmer discretion.
6. False. Each module and/or paragraph in a program should be restricted to a *single* function.
7. False. A flowchart is *procedural* in nature, while a hierarchy chart is *functional*.
8. True.
9. True.
10. True.

Exercises

1. New Master

100000000CALDWELL	CHI0206	A350000981000000000
111111111SMITH	DET0205	A180000781000000000
222222222JONES	CHI0404	E230000381200000980
444444444MILGROM	DET0207	G280000481240000480
555555555CRAWFORD	WAS0305	E350000981300000581

Error Messages

ERROR — NO CORRESPONDING MASTER RECORD FOR 666666666

Deleted File

333333333BAKER	BOS0306P195000182180001280
----------------	----------------------------

2. *Debugging*: The incorrect salary update is easily traced to the paragraph UPDATE-EMPLOYEE-SALARY. The problem is that the MOVE statements of lines 1625-1628 are out of order; specifically, line 1628 should *precede* line 1625.

The title for Wicks, social security number 400-00-0000, is wrong. This record was added to the master file, and consequently the problem may be in the addition routine, lines 1721-1730. Sure enough, there is no statement to move TRANS-TITLE-CODE to NEW-TITLE-CODE (line 1724 was deleted from the correct program of the chapter).

The record with social security number 333-33-3333 is missing from the new master. The logic in 020-COMPARE-IDS appears correct, as do all state-

ments associated with WS-OLD-MAST-READ-SWITCH. We turn our attention to the transaction for the missing record, a correction, and to the paragraph 080-CORRECT-OLD-RECORD. Note well that there is no WRITE statement at the end of the paragraph, and hence any corrected records will disappear.

Observe also that record 777-77-7777 was not added to the master file, nor was it flagged in an error message. It too is missing, but for a different reason. The PERFORM statement of lines 1020-1040 indicates that processing will continue until *either*, rather than *both*, the old master and transaction files are empty. The solution — change OR to AND in line 1040.

CHAPTER 12

True/False

1. True.
2. True.
3. True.
4. True.
5. False. Inactive records are removed with the DELETE verb.
6. False. Nonsequential means nonsequential!!
7. False. *Unique* keys are required, else the indexed file will not be properly established.
8. True.
9. False. The record key must be defined within the indexed file and be unique for each record.
10. False. Records are added by writing them to the file.

Exercises

1. *Debugging*: The concept of a hierarchy chart and the associated idea of *functional* paragraphs is very useful in debugging this program. Examination of the output reveals problems in deletion, correction and salary modification. Accordingly, one has to focus only on the paragraph in question and the difficulties become readily apparent.

Consider first the problem of deletions — Crawford, social security number 555-55-5555, should have been deleted from the indexed file. The record was in fact written to the file of deleted records, but remained in the indexed file. The problem is that a DELETE statement is *missing* from the paragraph 090-DELETE-NDX-RECORD. Recall that indexed records must be *explicitly* deleted, whereas a sequential update implicitly deletes records by not writing them to the new master.

Failure to correct performance is attributable to the paragraph 080-CORRECT-NDX-RECORD. Our attention is first directed to lines 1610-1620 which examine the transaction performance code. Apparently, there is nothing wrong with this statement, so what then is the problem? *Careful* examination reveals that a period is missing from line 1600; hence, lines 1590-1620 constitute a single nested IF statement. In other words, lines 1610 and 1620 are executed *only* if TRANS-EDUCATION-CODE is not blank. The solution is to insert a period at the end of line 1600.

The difficulty with salary updates leads one to the paragraph 075-UPDATE-EMPLOYEE-SALARY. After careful study, the reader may conclude that this paragraph is correct, so that the problem lies elsewhere, but where? The experienced programmer must have the ability to know when a dead end is reached, and to try another approach. A logical thought is that if there is nothing wrong with the logic to do a salary update, then perhaps there is a problem

in calling this routine. Lines 1280-1290 show that 075-UPDATE-EMPLOYEE-SALARY is called when the 88-level entry SALARY-UPDATE is satisfied. Checking further, we find that the definition of SALARY-UPDATE in line 550 is wrong. Recall from the problem specifications and test data that a transaction code of U rather than X implies a salary update.

Finally, the duplicate add message for the last transaction record must be accounted for. We see that the last transaction, social security number 777-77-7777, is an addition, and that the record is correctly added to the new master. The invalid error message results when the last transaction is processed a second time. Recall that a nonsequential update is driven by the transaction file. The correct program structure is an *initial* read, with a second read as the *last* statement of the performed routine. Examination of Figure 12.12 shows that the statement to read the transaction file is incorrectly placed. Move line 1125 to appear before the PERFORM of line 1030 and after line 1200.

B

**RESERVED
WORD LIST**

The following is a list of TRS-80 reserved words where:

* denotes reserved words not reserved in ANSI standard COBOL

+ denotes ANSI COBOL reserved words not reserved by the compiler. Their appearance will generate a warning at the end of the compilation listing.

** denotes system-name.

ACCEPT	ALPHABETIC	AREA
ACCESS	+ALSO	+AREAS
ADD	ALTER	+ASCENDING
ADVANCING	ALTERNATE	ASSIGN
AFTER	AND	AT
ALL	ARE	AUTHOR
*BEEP	*BLINK	BY
BEFORE	BLOCK	
BLANK	+BOTTOM	
CALL	+CODE-SET	COMPUTE
+CANCEL	COLLATING	CONFIGURATION
+CD	+COLUMN	CONTAINS
+CF	COMMA	+CONTROL
+CH	+COMMUNICATION	+CONTROLS
CHARACTER	COMP	*CONVERT
CHARACTERS	*COMP-1	COPY
+CLOCK-UNITS	*COMP-3	CORR
CLOSE	COMPUTATIONAL	CORRESPONDING
+COBOL	*COMPUTATIONAL-1	+COUNT
+CODE	*COMPUTATIONAL-3	CURRENCY
DATA	+DEBUG-SUB-1	+DESCENDING
DATE	+DEBUG-SUB-2	+DESTINATION
+DATE-COMPILED	+DEBUG-SUB-3	+DETAIL
DATE-WRITTEN	+DEBUGGING	+DISABLE
DAY	DECIMAL-POINT	DISPLAY
+DE	DECLARATIVES	DIVIDE
+DEBUG-CONTENTS	DELETE	DIVISION
+DEBUG-ITEM	+DELIMITED	DOWN
+DEBUG-LINE	+DELIMITER	DUPLICATES
+DEBUG-NAME	DEPENDING	DYNAMIC
*ECHO	+END-OF-PAGE	ERROR
+EGI	+ENTER	+ESI
ELSE	ENVIRONMENT	+EVERY
+EMI	+EOP	EXCEPTION

+ENABLE END	EQUAL *ERASE	EXIT EXTEND
FD FILE FILE-CONTROL	FILLER +FINAL FIRST	+FOOTING FOR FROM
+GENERATE GIVING	GO GREATER	+GROUP
+HEADING *HIGH	HIGH-VALUE HIGH-VALUES	
I-O I-O-CONTROL IDENTIFICATION IF IN INDEX	INDEXED +INDICATE INITIAL +INITIATE INPUT INPUT-OUTPUT	INSPECT INSTALLATION INTO INVALID IS
JUST	JUSTIFIED	
KEY		
LABEL +LAST LEADING LEFT +LENGTH LESS	+LIMIT +LIMITS +LINAGE +LINAGE-COUNTER LINE +LINE-COUNTER	LINES LINKAGE LOCK LOW LOW-VALUE LOW-VALUES
MEMORY +MERGE +MESSAGE	MODE MODULES MOVE	+MULTIPLE MULTIPLY
NATIVE +NEGATIVE NEXT	NO NOT +NUMBER	NUMERIC
OBJECT-COMPUTER OCCURS OF OFF	OMITTED ON OPEN +OPTIONAL	OR ORGANIZATION OUTPUT +OVERFLOW

PAGE
+PAGE-COUNTER
PERFORM
+PF
+PH
PIC
PICTURE

+PLUS
+POINTER
POSITION
+POSITIVE
*PRINT
+PRINTING
PROCEDURE

+PROCEDURES
PROCEED
PROGRAM
PROGRAM-ID
*PROMPT

+QUEUE

QUOTE

QUOTES

RANDOM
+RD
READ
+RECEIVE
RECORD
RECORDS
REDEFINES
REEL
+REFERENCES
RELATIVE
+RELEASE

+REMAINDER
+REMOVAL
RENAMES
REPLACING
+REPORT
+REPORTING
+REPORTS
+RERUN
+RESERVE
+RESET
+RETURN

*REVERSE
+REVERSED
REWIND
REWRITE
+RF
+RH
RIGHT
ROUNDED
RUN

SAME
+SD
+SEARCH
SECTION
SECURITY
+SEGMENT
+SEGMENT-LIMIT
SELECT
+SEND
SENTENCE
SEPARATE
SEQUENCE
SEQUENTIAL
SET
SIGN

SIZE
+SORT
+SORT-MERGE
+SOURCE
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STANDARD
STANDARD-1
START
STATUS
STOP
+STRING
+SUB-QUEUE-1

+SUB-QUEUE-2
+SUB-QUEUE-3
SUBTRACT
+SUM
+SUPPRESS
**SWITCH-1
**SWITCH-2
,
,
,
**SWITCH-8
+SYMBOLIC
SYNC
SYNCHRONIZED

*TAB
+TABLE
TALLYING
+TAPE
+TERMINAL
+TERMINATE

+TEXT
THAN
THROUGH
THRU
TIME
TIMES

TO
+TOP
TRAILING
+TYPE

UNIT
*UNLOCK
+UNSTRING

UNTIL
UP
+UPON

USAGE
USE
USING

VALUE

VALUES

VARYING

WHEN
WITH

WORDS
WORKING-STORAGE

WRITE

ZERO

ZEROES

ZEROS

+

-

=

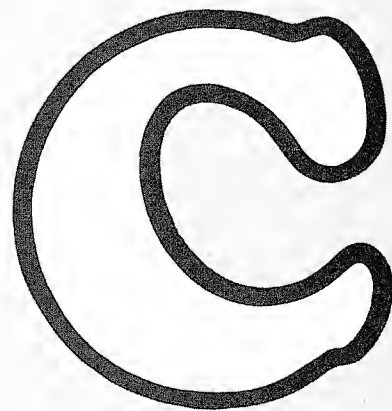
>

<

*

/

**



**TRS-80 COBOL
SYNTAX**

The TRS-80 COBOL language is based upon the ANSI X3.23-1974 COBOL standard. Minor departures from that document are reflected in the syntax description which follows but are not separately noted. Semantic rules are not changed.

The description is in a condensed form of the standard COBOL syntax notation. In some cases separate formats are combined and general terms are employed for user names.

System-names and implementation restrictions are:

computer-name:	User-defined word
program-name:	8-character name
switch-names:	SWITCH-1, . . . , SWITCH-8
device-types:	PRINT INPUT OUTPUT INPUT-OUTPUT RANDOM
external-file-name:	One- to thirty-character name

IDENTIFICATION DIVISION GENERAL FORMAT

IDENTIFICATION DIVISION.

PROGRAM-ID. program-name.

[AUTHOR. [comment-entry] . . .]

[INSTALLATION. [comment-entry] . . .]

[DATE-WRITTEN. [comment-entry] . . .]

[SECURITY. [comment-entry] . . .]

ENVIRONMENT DIVISION GENERAL FORMAT

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER. computer-name.

OBJECT-COMPUTER. computer-name

[, MEMORY SIZE integer {WORDS }]
 {CHARACTERS}
 {MODULES }

[, PROGRAM COLLATING SEQUENCE IS alphabet-name].

[SPECIAL-NAMES. [, switch-name

 {ON STATUS IS condition-name-1 [, OFF STATUS IS condition-name-2}}]
 -- --

 {OFF STATUS IS condition-name-2 [, ON STATUS IS condition-name-1}}]
 -- --

[, alphabet-name IS {STANDARD-1}} ...
 {NATIVE }

[, CURRENCY SIGN IS literal-1]
 -- --

[, DECIMAL-POINT IS COMMA].]
 ----- -- -----

[INPUT-OUTPUT SECTION.

FILE-CONTROL.

 {file-control-entry} ...

[I-O-CONTROL.

 [; SAME AREA FOR file-name-1 [, file-name-2] ...]. ...]]

FILE CONTROL ENTRY GENERAL FORMAT

FORMAT 1

SELECT file-name

ASSIGN TO device-type {"external-file-name"}
----- {data-name-1 }

[; ORGANIZATION IS SEQUENTIAL]

[; ACCESS MODE IS SEQUENTIAL]

[; FILE STATUS IS data-name-2].

FORMAT 2

SELECT file-name

ASSIGN TO RANDOM, {"external-file-name"}
----- {data-name-1 }

; ORGANIZATION IS RELATIVE

[; ACCESS MODE IS { SEQUENTIAL [; RELATIVE KEY IS data-name-2]}]
----- {RANDOM } , RELATIVE KEY IS data-name-2 }
----- {{DYNAMIC} }

[; FILE STATUS IS data-name-3].

FORMAT 3

SELECT file-name

ASSIGN TO RANDOM, {"external-file-name"}
----- {data-name-1 }

; ORGANIZATION IS INDEXED

```

[; ACCESS MODE IS {SEQUENTIAL}]
-----
                {RANDOM      }
                -----
                {DYNAMIC    }
                -----

; RECORD KEY IS data-name-2
-----

[; ALTERNATE RECORD KEY IS data-name-3 [WITH DUPLICATES]]...
-----

[; FILE STATUS IS data-name-4].
-----

```

DATA DIVISION GENERAL FORMAT

DATA DIVISION.

[FILE SECTION.

[FD file-name

--

```

[; BLOCK CONTAINS [integer-1 TO] integer-2 {RECORDS  }
-----
                                     {CHARACTERS}

```

```

[; RECORD CONTAINS [integer-3 TO] integer-4 CHARACTERS]
-----

```

```

; LABEL {RECORD IS  } {STANDARD}
-----
      {RECORDS ARE} {OMITTED }
-----

```

```

[; VALUE OF LABEL IS nonnumeric-literal-1]
-----

```

```

[; DATA {RECORD IS  } data-name-1 [, data-name-2] ... ]
-----
      {RECORDS ARE}
-----

```

[record-description-entry] ...] ...

[WORKING-STORAGE SECTION.

```

[77-level-description-entry] ... ]
[record-description-entry ]

```

— — — — —

```
[77-level-description-entry] ... ]]
```

FORMAT 1

```
level-number {data-name-1}
```

```
[; REDEFINES data-name-2]
```

```
[; {PICTURE} IS character-string]
```

{PIC }

```
[; [USAGE IS] {COMPUTATIONAL }]
```

{COMP }

{COMPUTATIONAL-1}

{COMP-1 }

{COMPUTATIONAL-3}

{COMP-3 }

```
{DISPLAY      }
```

{INDEX }

[; [SIGN IS] TRAILING [SEPARATE CHARACTER] 1

```
[ ] OCCURS {integer-1 TIMES
----- {integer-1 TO integer-2 TIMES DEPENDING ON data-name-3}
```

[INDEXED BY index-name-1 [, index-name-2] ...]]

[; {SYNCHRONIZED} [LEFT]]

```
{SYNC      } [RIGHT]
```

```

[; {JUSTIFIED} RIGHT]
-----
{JUST      }
-----

[; BLANK WHEN ZERO]
-----

[; VALUE IS literal]
-----

```

FORMAT 2

```

66 data-name-1; RENAMES data-name-2 [{THROUGH} data-name-3].
-----
                                {THRU  }
                                -----

```

FORMAT 3

```

88 condition-name; {VALUE IS }
-----
                   {VALUES ARE}
                   -----

literal-1 [{THROUGH} literal-2]
-----
                   {THRU  }
                   -----

[, literal-3 [{THROUGH} literal-4] ] ...
-----
                   {THRU  }
                   -----

```

PROCEDURE DIVISION GENERAL FORMAT

FORMAT 1

```

PROCEDURE DIVISION [USING data-name-1 [, data-name-2] ... ] .
-----

[DECLARATIVES.
-----

{section-name SECTION [segment-number]. declarative-sentence
-----

```

[paragraph-name. [sentence] ...] ... } ...

END DECLARATIVES.]

{section-name SECTION [segment-number].

[paragraph-name. [sentence] ...] ... } ...

END PROGRAM.

FORMAT 2

PROCEDURE DIVISION [USING data-name-1 [, data-name-2] ...] .

{paragraph-name. [sentence] ... } ...

END PROGRAM.

GENERAL FORMAT FOR VERBS

ACCEPT {identifier-1 [, UNIT {identifier-2}]
----- ---- {literal-1 }

[, LINE {identifier-3}] [, POSITION {identifier-4}]
---- {literal-2 } ----- {literal-3 }

[, SIZE {identifier-5}] [, PROMPT {literal-5}]
---- {literal-4 } -----

[, ECHO] [, CONVERT] [, TAB] [, ERASE] [, NO BEEP]
----- --- ----- -- ----

[, {OFF}] [, ON EXCEPTION identifier-6 imperative statement]].
---- -- -----

ACCEPT identifier FROM {DATE}

{DAY }

{TIME}

ADD {identifier-1} [, identifier-2] ... TO identifier-m [ROUNDED]
--- {literal-1 } [, literal-2] -- -----

[, ON SIZE ERROR imperative-statement]

```

ADD {identifier-1}, {identifier-2} [, {identifier-3} ...
--- {literal-1 } {literal-2 } [, {literal-3 } ]

    GIVING identifier-m [ROUNDED]
    -----

    [, ON SIZE ERROR imperative-statement]
    -----

ADD {CORRESPONDING} identifier-1 TO identifier-2
--- -----
{CORR }
-----

    [ROUNDED] [, ON SIZE ERROR imperative-statement]
    -----

ALTER procedure-name-1 TO [PROCEED TO] procedure-name-2
-----

    [, procedure-name-3 TO [PROCEED TO] procedure-name-4] ...
    -----

CALL {identifier-1} [USING data-name-1 [, data-name-2] ... ]
---- {literal-1 } -----

CLOSE file-name-1 [{REEL} [WITH NO REWIND] ]
-----
    {UNIT}
    -----
    WITH {NO REWIND}
    -----
    {LOCK }
    -----

    [, file-name-2 [{REEL} [WITH NO REWIND] ] ] ...
    -----
    {UNIT}
    -----
    WITH {NO REWIND}
    -----
    {LOCK }
    -----

COMPUTE identifier-1 [ROUNDED] = arithmetic-expression
-----

    [, ON SIZE ERROR imperative-statement]
    -----

DELETE file-name RECORD [, INVALID KEY imperative-statement]
-----

DISPLAY {{identifier-1} [, UNIT {identifier-2} ]
----- {literal-1 } ----- {literal-2 }

    [, LINE {identifier-3}][, POSITION {identifier-4}]
    ----- {literal-3 } ----- {literal-4 }

```



```

[, SIZE {identifier-5}][, BEEP][, ERASE]
---- {literal-5 } ----

[, {HIGH}][, BLINK][, REVERSE]} ...
----

{LOW }
---

DIVIDE {identifier-1} INTO identifier-2 [ROUNDED]
----- {literal-1 } ----

      [, ON SIZE ERROR imperative-statement]
      ----

DIVIDE {identifier-1} INTO {identifier-2} GIVING identifier-3
----- {literal-1 } ---- {literal-2 } -----

      [ROUNDED] [, ON SIZE ERROR imperative-statement]
      -----

DIVIDE {identifier-1} BY {identifier-2} GIVING identifier-3 [ROUNDED]
----- {literal-1 } -- {literal-2 } -----

      [, ON SIZE ERROR imperative-statement]
      ----

EXIT [PROGRAM].
-----

GO TO procedure-name-1
--

GO TO procedure-name-1 [, procedure-name-2] ... , procedure-name-n
--

      DEPENDING ON identifier
      -----

IF condition; {statement-1 } {; ELSE statement-2 }
--
      {NEXT SENTENCE} {; ELSE NEXT SENTENCE}
      -----

INSPECT identifier-1
-----

      [TALLYING identifier-2 FOR {{ALL } {identifier-3}}
      -----
      {literal-1 }}
      {{LEADING}
      -----
      { CHARACTERS }
      -----

      [{BEFORE} INITIAL {identifier-4}]]
      -----
      {literal-2 }
      {AFTER }
      -----

```

```

[REPLACING      {{ALL      } {identifier-5}} BY {identifier-6}
-----      ---      {literal-3  } -- {literal-4  }
                  {{LEADING}                      }
                  -----
                  {{FIRST  }                      }
                  -----
                  {      CHARACTERS                }
                  -----

      [{BEFORE}  INITIAL {identifier-7}]]
      -----      {literal-5  }
      {AFTER }
      -----

```

NOTE: The TALLYING option, the REPLACING option, or both options must be selected.

```

MOVE {identifier-1} TO identifier-2 [, identifier-3]...
---- {literal      } --

```

```

MOVE {CORRESPONDING} identifier-1 TO identifier-2
-----
      {CORR          }
      -----

```

```

MULTIPLY {identifier-1} BY identifier-2 [ROUNDED]
----- {literal-1  } -- -----

```

```

      [, ON SIZE ERROR imperative-statement]
      -----

```

```

MULTIPLY {identifier-1} BY {identifier-2} GIVING identifier-3
----- {literal-1  } -- {literal-2  } -----

```

```

      [ROUNDED] [, ON SIZE ERROR imperative-statement]
      -----

```

```

OPEN {{INPUT file-name-1 [WITH NO REWIND]}
-----      -----
      [, file-name-2 [WITH NO REWIND]]...
      -----

```

```

      {OUTPUT file-name-3 [WITH NO REWIND]}
      -----
      [, file-name-4 [WITH NO REWIND]]...
      -----

```

```

      {I-O file-name-5}[, file-name-6]...
      -----

```

```

      {EXTEND file-name-7}[, file-name-8]...}...
      -----

```

```

PERFORM procedure-name-1 [{THROUGH} procedure-name-2]
-----
                  {THRU  }
                  -----

```

```

PERFORM procedure-name-1 [{THROUGH} procedure-name-2]
-----
                        {THRU   }
                        -----
        {identifier-1} TIMES
        {literal-1   } -----

PERFORM procedure-name-1 [{THROUGH} procedure-name-2]
-----
                        {THRU   }
                        -----
        UNTIL condition-1
        -----

PERFORM procedure-name-1 [{THROUGH} procedure-name-2]
-----
                        {THRU   }
                        -----

        VARYING {identifier-2} FROM {identifier-3}
        ----- {index-name-1} ---- {index-name-2}
                                   {literal-1   }

        BY {identifier-4} UNTIL condition-1
        -- {literal-3   } -----

        [AFTER {identifier-5} FROM {identifier-6}
        ----- {index-name-3} ---- {index-name-4}
                                   {literal-3   }

        BY {identifier-7} UNTIL condition-2
        -- {literal-4   } -----

        [AFTER {identifier-8} FROM {identifier-9}
        ----- {index-name-5} ---- {index-name-6}
                                   {literal-5   }

        BY {identifier-10} UNTIL condition-3 ] ]
        -- {literal-6   } -----

READ file-name RECORD [INTO identifier]
----
        [; AT END imperative-statement]
        ---

READ file-name [NEXT] RECORD [WITH NO LOCK] [INTO identifier]
----
        [; AT END imperative-statement]
        ---

READ file-name RECORD [WITH NO LOCK] [INTO identifier]
----
        [; KEY IS data-name]
        ---
        [; INVALID KEY imperative-statement]
        -----

REWRITE record-name [FROM identifier]
-----
        [; INVALID KEY imperative-statement]
        -----

```

```

SET {identifier-1 [, identifier-2] ...} TO {identifier-3}
--- {index-name-1 [, index-name-2] ...} -- {index-name-3}
                                         {integer-1 }

SET index-name-4 [, index-name-5] ... {UP BY } {identifier-4}
---                                     -- -- {integer-2 }
                                         {DOWN BY}

START file-name [KEY {IS EQUAL TO      } data-name]
-----
                        {IS =              }
                        {IS GREATER THAN }
                        -----
                        {IS >              }
                        {IS NOT LESS THAN}
                        --- ----
                        {IS NOT <         }
                        ---

[; INVALID KEY imperative-statement]
-----

STOP {RUN          }
----
    {literal }

SUBTRACT {identifier-1} [, identifier-2] ... FROM identifier-m
----- {literal-1 } [, literal-2 ] ----

    [ROUNDED] [; ON SIZE ERROR imperative-statement]
    -----

SUBTRACT {identifier-1} [, identifier-2] ... FROM {identifier-m}
----- {literal-1 } [, literal-2 ] ---- {literal-m }

    GIVING identifier-n [ROUNDED]
    -----

    [; ON SIZE ERROR imperative-statement]
    -----

SUBTRACT {CORRESPONDING} identifier-1 FROM identifier-2 [ROUNDED]
-----
    {CORR          }
    -----
    [; ON SIZE ERROR imperative-statement]
    -----

UNLOCK file-name-1 RECORD
-----

USE AFTER STANDARD {EXCEPTION}
---
    {ERROR }
    -----
    PROCEDURE ON {file-name-1 [, file-name-2] ...} .
    -----

```

```

        {INPUT                                }
        -----
        {OUTPUT                                }
        -----
        {I-O                                  }
        ---
        {EXTEND                               }
        -----

```

```

WRITE record-name [FROM identifier-1]
-----

```

```

    {BEFORE} ADVANCING {{identifier-2} {LINE }}
    -----            {{integer      } {LINES}}
    {AFTER }           {          PAGE    }
    -----            -----

```

```

WRITE record-name [FROM identifier]
-----

```

```

    [; INVALID KEY imperative-statement]
    -----

```

GENERAL FORMAT FOR CONDITIONS

RELATION CONDITION:

```

{identifier-1 } {IS [NOT] GREATER THAN} {identifier-2 }
{literal-1   } {IS [NOT] GREATER THAN} {literal-2   }
{index-name-1 } {IS [NOT] LESS THAN   } {index-name-2 }
               {IS [NOT] EQUAL TO     }
               {IS [NOT] >              }
               {IS [NOT] <              }
               {IS [NOT] =              }

```

CLASS CONDITION:

```

    identifier IS [NOT] {NUMERIC   }
                       -----
                       {ALPHABETIC}
                       -----

```

CONDITION-NAME CONDITION:

```

    condition-name

```

SWITCH-STATUS CONDITION:

condition-name

NEGATED SIMPLE CONDITION:

NOT simple-condition

COMBINED CONDITION:

condition {{AND} condition} ...

{OR }
--

MISCELLANEOUS FORMATS

QUALIFICATION:

{data-name-1 } [{OF} data-name-2] ...
{condition-name} --
 {IN}
 --

paragraph-name [{OF} section-name]
 --
 {IN}
 --

SUBSCRIPTING:

{data-name } (subscript-1 [, subscript-2 [, subscript-3]])
{condition-name}

INDEXING:

{data-name } ({index-name-1 [{+} literal-2]}
{condition-name} {literal-1 {-} }

[, {index-name-2 [{+} literal-4]}
 {literal-3 {-} }

[, {index-name-3 [{+} literal-6] }]])
 {literal-5 {-} }

IDENTIFIER:

FORMAT 1

```
data-name-1 [{OF} data-name-2] ...  
  --  
  {IN}  
  --  
  
  [(subscript-1 [, subscript-2 [, subscript-3] ] ) ]
```

FORMAT 2

```
data-name-1 [{OF} data-name-2] ... [( {index-name-1 [{+} literal-2]  
  --                               {literal-1      {-}  
  {IN}  
  --  
  
  [, {index-name-2 [{+} literal-4}]  
    {literal-3      {-}          }  
  
  [, {index-name-3 [{+} literal-6}] ]]]  
    {literal-5      {-}          }
```

GENERAL FORMAT FOR COPY STATEMENT

COPY text-name

D

GLOSSARY

A-Margin: Columns 8-11 on a coding pad. Many COBOL elements are *required* to begin in the A-Margin; e.g., division, section, and paragraph headers; 01 and 77 level entries, FDs, etc.

Access Mode: The manner in which records are to be operated upon within a file; the text discussed *sequential* and *indexed sequential* files.

Actual Decimal Point: The physical (as opposed to the assumed or implied) representation of the decimal point position in a data item.

Alphanumeric Character: Any character in the computer's character set.

Arithmetic Operator: One of four symbols denoting arithmetic: + (addition), - (subtraction), * (multiplication), and / (division).

Assumed Decimal Point: A decimal point position with logical meaning, but not physical representation, and which does not have an actual character in a data item. The assumed decimal point is denoted by a V.

AT END Condition: A condition caused during the execution of a READ statement for a sequentially accessed file; i.e., a file with N records will be read N+1 times in order that the AT END condition may be detected.

B-Margin: Columns 12-72 on a coding pad. Any COBOL element which is not required to begin in the A-Margin, must begin in the B-Margin.

Called Program: see Subprogram.

Calling Program: A program which executes a CALL to another program.

COBOL Character Set: The complete COBOL character set consists of the 51 characters listed below.

<i>Character</i>	<i>Meaning</i>
0, 1, . . . , 9	digit
A, B, , Z	letter
	space (blank)
+	plus sign
-	minus sign (hyphen)
*	asterisk
/	stroke (virgule, slash)
=	equal sign
\$	currency sign
,	comma (decimal point)
;	semicolon
.	period (decimal point)
"	quotation mark
(left parenthesis
)	right parenthesis
>	greater than symbol
<	less than symbol

Coding Standards: Additional requirements imposed on a program by an installation to enhance a program's readability (see Chapter 7).

Compound Condition: A condition which connects two or more conditions with the 'AND' or the 'OR' logical operator.

Comment Line: A source program line denoted by an asterisk in column 7, and which serves only for documentation.

Compiler: A *program* which translates a higher level language, e.g., COBOL, into a lower level machine language. The COBOL compiler on the TRS-80 is invoked by the command RSCOBOL.

Compile-Time: The time at which a COBOL source program is translated, by a COBOL compiler, to a COBOL object program.

Condition-Name (88 level entry): A user-defined word assigned to a specific value or set of values (see Chapter 6).

Conditional Expression: A simple or compound condition specified in an IF or PERFORM statement.

CONFIGURATION SECTION: A section of the Environment Division that describes overall specifications of source and object computers.

Control Break: A change in a designated field (see Chapter 8).

DATA DIVISION: The third major component of a COBOL program. The FILE SECTION describes the files used by a program and the records contained within. The WORKING-STORAGE SECTION describes additional records and/or data names required by the program which are not present in a file.

Debugging: The process of finding and correcting errors in a program (see Chapter 5).

DECLARATIVES: A set of one or more special purpose sections, written at the beginning of the Procedure Division, the first of which is preceded by the key word DECLARATIVES and the last of which is followed by the key words END DECLARATIVES (see Chapter 12).

Editing Character: A single or fixed two-character combination as follows (see Chapter 8):

<i>Character</i>	<i>Meaning</i>
B	space
0	zero
+	plus
-	minus
CR	credit
DB	debit
Z	zero suppression
*	check protection
\$	currency sign
,	comma
.	period (decimal point)
/	slash

Elementary Item: A data name which is not further subdivided, and which *always* has a picture clause in its definition.

ENVIRONMENT DIVISION: The second major component of a COBOL program. It is divided into a CONFIGURATION SECTION and an INPUT-OUTPUT SECTION. The latter is the more important and ties programmer-chosen file names to system names.

Execution Time: see Object Time.

Field: One or more characters which describe an attribute; e.g., name, social security number, salary, etc.

Figurative Constant: A compiler-generated value referenced through the use of certain reserved words, e.g., SPACES or ZEROS.

File: A collection of records.

FILE-CONTROL: The name of the Environment Division paragraph in which the data files for a source program are declared. The FILE-CONTROL paragraph consists of SELECT statements which tie programmer-chosen file names to physical files.

FILE SECTION: The section of the Data Division that contains file description entries together with their associated record descriptions.

Flowchart: A pictorial representation of the logic in a program.

Group Item: A data name which is further subdivided; a group item *never* has a picture clause.

Hierarchy Chart: A program's "organization chart", analogous to a corporate organization chart, which graphically depicts the relationship of COBOL paragraphs to each other. Each level of a hierarchy chart corresponds to a COBOL PERFORM statement.

IDENTIFICATION DIVISION: The first major component of a COBOL program. It must contain the PROGRAM-ID paragraph and may contain additional documentation such as author, installation, etc.

Identifier: A data-name, followed, as required, by a combination of qualifiers, subscripts, and indices necessary to make unique reference to a data item.

Imperative Statement: A statement beginning with a verb, and specifying an unconditional action to be taken.

Index: A data item, the contents of which represent the identification of a particular element in a table (see Chapter 10).

Indexed File: A type of file organization which allows individual records to be accessed either sequentially or nonsequentially (see Chapter 12).

INPUT-OUTPUT SECTION: The section of the Environment Division containing the FILE-CONTROL paragraph that names the files and the external media required by an object program.

INVALID KEY Condition: A condition caused when a specific value of the key associated with an indexed file is determined to be invalid, e.g., when a particular key does not exist in an indexed file (see Chapter 12).

Level Number: Numbers (01 through 49) which are used to indicate the relative position of data names within a record. Level numbers 77 and 88 have special significance.

LINKAGE SECTION: The section in the Data Division of the *called* program that describes the data items available from the calling program. These data items may be referred to by both the calling and called program (see Chapter 9).

Literal: A quantity with an unchanging value; may be *numeric* (i.e., a constant), or *non-numeric* (a character string enclosed in quotes).

Logical Operator: One of the reserved words AND, OR, or NOT.

Non-numeric Literal: A character-string bounded by quotation marks. The string of characters may include any character in the computer's character set.

Numeric Literal: A literal composed of one or more numeric characters, that may contain a decimal point, an algebraic sign, or both.

OBJECT-COMPUTER: An Environment Division paragraph which describes the computer environment, in which the object program is executed.

Object Program: The result of the operation of a COBOL compiler on a source program.

Object Time: The time at which an object program is executed.

Paragraph: One or more COBOL sentences.

PICTURE Clause: A Data Division entry used to describe the *length* and *type* of a data name.

Procedure: A paragraph or section.

PROCEDURE DIVISION: The fourth and final component of a COBOL program. It contains a program's logic and may be divided into sections or paragraphs.

Programmer-supplied name: A file, data, or procedure (paragraph or section) name made up by the programmer according to COBOL rules.

Pseudocode: "Neat notes" to oneself which convey the logic in a program. Pseudocode is limited to the three basic building blocks of structured programming, but otherwise has no formal syntax.

Punctuation Character: A character that belongs to the following set: , (comma), ; (semicolon), . (period), " (quotation mark), ((left parenthesis),) and (right parenthesis).

Qualified Data-Name: An identifier that is composed of a data-name followed by either of the connectives OF or IN, followed by a data-name qualifier (see Chapter 6).

Record: A set of facts (i.e., fields) about a logical entity.

Reference Format: A format that provides a standard method for describing COBOL source programs (see Appendix C).

Relational Operator: A reserved word(s) or a relational character used in the construction of a relational condition. The permissible operators and their meanings are:

<i>Relational Operator</i>	<i>Meaning</i>
IS (NOT) GREATER THAN	Greater than or not
IS (NOT) >	greater than
IS (NOT) LESS THAN	Less than or not
IS (NOT) <	less than
IS (NOT) EQUAL TO	Equal to or not
IS (NOT) =	equal to

Reserved Word: A predefined word, having specific meaning to the COBOL compiler, which must be used in a prescribed manner (see Appendix B).

Section: One or more COBOL paragraphs.

Sentence: A sequence of one or more statements, the last of which is terminated by a period followed by a space.

SOURCE-COMPUTER: An Environment Division paragraph which describes the computer environment in which the source program is compiled.

Source Program: A set of COBOL statements, beginning with an Identification Division and ending with the end of the Procedure Division, which is submitted to the compiler.

Special Character: A character that belongs to the following set:

<i>Character</i>	<i>Meaning</i>
+	plus sign
-	minus sign
*	asterisk
/	stroke (virgule, slash)
=	equal sign
\$	currency sign
,	comma
;	semicolon
.	period (decimal point)
''	quotation mark
(left parenthesis
)	right parenthesis
>	greater than symbol
<	less than symbol

Statement: A syntactically valid combination of words and symbols, beginning with a verb and written in the Procedure Division.

Structured Programming: A discipline which restricts the logic of any program to the three elementary building blocks of *sequence*, *selection*, and *iteration*. (Every program in this text is a structured program. See chapter 7).

Subprogram: A complete COBOL program which can be developed, compiled, and tested separately; also known as a called program (see Chapter 9).

Subscript: An integer whose value identifies a particular element in a table.

Symbol: A character with specific meaning; symbols may be relational, arithmetic, or for punctuation.

Table: A set of logically consecutive items of data that are defined in the Data Division by means of the OCCURS clause.

Top Down Approach: The philosophy of testing a program (or system) *before* it is completely finished. Top down development follows a program's hierarchy chart and requires the use of program *stubs* (see Chapters 11 and 12).

WORKING-STORAGE SECTION: The section of the Data Division describing data items not specified in input or output records.

INDEX

A

- as CEDIT command, 32, 34
- in PICTURE, 7, 44
- ACCEPT, 4, 8, 104-5, 117
- ACCESS IS RANDOM, 238
- ACCESS IS SEQUENTIAL, 238, 242
- ACCESS MODE, 238
- Actual decimal point, 146, 147, 298
- ADD, 48-49, 130
- AFTER ADVANCING, 54
- ALL, 61
- Alphabetic class test, 229
- Alphanumeric, 7, 44, 298
- A margin, 28, 34, 298
- American National Standards Institute (see ANSI)
- AND, 98-100
- ANSI, 13
- Arithmetic expression, 52
- Arithmetic symbols, 6, 52, 87
- ASSIGN (see SELECT)
- Assumed decimal point, 56-57, 64
- Asterisk:
 - in arithmetic expression, 52, 53
 - as comment, 34
 - in PICTURE, 146, 147
- AT END (see READ)
- AUTHOR, 43

B

- as CEDIT command, 32, 34
- in PICTURE, 109, 116, 146
- Backup, 215
- BACKUP utility, 215
- BEFORE ADVANCING, 54
- Binary search (see SEARCH ALL)
- Blank line, 129
- B margin, 28, 34, 298
- Braces, 42
- Brackets, 42
- Byte, 236

C

- as CEDIT command, 32
- CALL, 169-70
- Called program (see Subprogram)
- Calling program, 169-76
- CBL (as file extension), 26, 30
- CEDIT, 31-34, 168-69
- Character set, 298
- Check protection (see Asterisk)
- Class test (see IF, class test)
- CLOSE, 17, 54
- COB (as file extension), 26, 30
- Coding standards (see Standards)
- Column numbers in COBOL, 34, 75, 88
- Comma:
 - in editing, 57, 146, 147
 - in listing, 128

- COMMAND mode (see CEDIT)
- Comments, 34, 128-29, 298
- Compilation error, 70-81, 86-93
- Compiler, 24, 25, 298
- Compiler Options (see RSCOBOL)
- Completeness check, 230
- Compound test (see IF, compound test)
- COMPUTE, 52-53, 85, 106-7, 130
- Conditional expression, 298
- Conditional symbols, 6
- Condition name (see IF, condition name)
- CONFIGURATION SECTION, 7, 44
- Consistency check, 229
- Constants, 131 (see Literal)
- Continuation of alphanumeric literal, 132
- Control break, 148-65
- COPY, 168-69, 171, 176, 187, 191
- CORRESPONDING (see MOVE CORRESPONDING)
- CR:
 - in PICTURE, 146, 148, 151
- Cross reference, 27, 29
- Currency symbol (see Editing)

D

- as CEDIT command, 32
- DATA DIVISION, 3, 44-47

Dataname, 5, 127
 Dataname qualification (*see* Qualification)
 DATA RECORD IS, 46
 Data validation, 114
 DATE, 104
 Date check, 230
 DATE-COMPILED, 143
 DATE-WRITTEN, 43
 DAY, 104
 DB:
 in PICTURE, 146, 148
 Debugging, 69-96, 161-64
 (*see also* exercises in Chapters 8-12)
 Decimal alignment (*see* Editing)
 Decimal point (*see* Editing)
 DECLARATIVES, 240-41, 249, 298
 DELETE, 240
 DEPENDING ON (*see* OCCURS DEPENDING ON)
 DIR command, 30
 Direct access to table entries, 194
 DISPLAY, 4, 8, 104, 219
 DIVIDE, 50, 51, 130
 Documentation, 156, 218
 Dollar sign (*see* Editing)
 Double spacing (*see* AFTER ADVANCING)
 DO WHILE structure (*see* Iteration structure)
 DOWN BY (*see* SET)
 Duplicate datanames, 107-9

E

As CEDIT command, 32
 Editing, 56-57, 146-48, 160, 298
 EDIT mode (*see* CEDIT)
 Edit program, 217, 229-30
 Elementary item, 27, 45-46, 298
 ELSE, 58 (*see* IF)
 END DECLARATIVES, 240
 End-of-file condition, 13-14
 ENVIRONMENT DIVISION, 2, 16, 43, 77
 Error processing (*see* Edit program)
 Execution error, 82-85, 93-96, 161-64
 EXIT, 103
 EXIT PROGRAM, 169-70, 174, 175
 Exponentiation, 52
 Expression (*see* Arithmetic expression)

F

as CEDIT command, 32
 FD, 16, 46, 77, 87, 168
 Field, 12, 298
 Figurative constant, 47, 55, 298
 File, 12
 FILE-CONTROL, 44, 298
 File maintenance:
 nonsequential, 236-49
 sequential, 214-30
 File name:
 in COBOL, 5
 under TRSDOS, 26

FILE SECTION, 16, 46
 FILE STATUS, 239, 240, 245
 FILLER, 46
 Fixed length record, 185
 Floating dollar sign, 57, 147
 Floating minus sign, 147
 Floating plus sign, 147
 Flowchart, 12-14
 FORMS command, 30
 FROM (*see* SUBTRACT, WRITE)

G

GIVING (*see* Arithmetic verbs)
 GO TO, 103, 137
 Grauer, R., 214
 Group item, 27, 45-46, 298

H

Hard copy, 30
 Heading line, 118
 Hierarchy:
 of arithmetic operations, 52
 between group and elementary items, 45
 chart, 4, 63-64, 138, 156, 218-19, 245, 298
 in IF statement, 99
 HOLD, 27, 30
 Housekeeping, 13, 14
 Hyphen, 5, 87, 132

I

as CEDIT command, 32, 34
 IDENTIFICATION DIVISION, 2, 43
 Identifier, 298
 IF, 58
 compound test, 99-100, 135, 298
 condition name, 100-101, 115, 135, 151, 216
 indentation, 102
 nested, 101-2, 117, 130
 notation, 42, 43
 period, 6, 17
 Imperative statement, 298
 Implied conditions (*see* IF, implied conditions)
 Implied decimal point (*see* Assumed decimal point)
 IN (*see* Qualification)
 Indentation in COBOL, 130-31, 143
 Index, 185, 188-89
 INDEXED BY, 184-85
 Indexed files, 236-40
 Initial read, 17, 18, 19, 59, 85
 INPUT (*see* OPEN)
 INPUT-OUTPUT SECTION, 16, 44
 Input validation (*see* Data validation)
 INSERT mode (*see* CEDIT)
 INSPECT, 109, 117, 192
 INSTALLATION, 43
 Installation standards (*see* Standards)

Instruction explosion, 25
 INVALID KEY, 239-40, 242, 245
 Iteration structure, 136-37

K

KILL command, 30, 31, 215

L

as CEDIT command, 32
 as compiler option, 27
 LABEL RECORDS, 46
 Labels, 16
 Level number, 7, 45-46, 298
 Limit check, 229
 LINE, 104, 105
 LINES (*see* WRITE)
 LINKAGE SECTION, 169-71, 175
 Literal, 5-6, 47, 55, 132-33, 298
 Loading a table (*see* Table)
 Logical operator, 298
 Logic error, 82-86
 LST (as file extension), 26, 30, 242

M

Machine language, 24-26, 27, 188
 Maintenance (*see* File maintenance)
 Minus sign:
 in arithmetic expression, 52
 in PICTURE, 146, 147
 MOVE, 54-57
 MOVE CORRESPONDING, 107-9
 Multilevel table (*see* Table)
 MULTIPLY, 50, 51, 130

N

as CEDIT command, 32, 34
 Negative data (*see* Signed number)
 Nested IF (*see* IF)
 New Master (*see* File maintenance)
 NEXT SENTENCE, 101
 Non-numeric literal (*see* Literal)
 Nonsequential file maintenance (*see* File maintenance)
 NOT, 99
 Notation in COBOL statements, 42, 282-96
 NUMERIC, 229
 Numeric class test, 229
 Numeric literal (*see* Literal)

O

as compiler option, 27
 OBJECT-COMPUTER, 7, 44
 Object language, 24
 Object module, 171
 OCCURS, 110-11, 184-85, 186, 190
 with three level table, 202-3
 with two level table, 197, 199

OCCURS DEPENDING ON, 184-85
 OF (*see* Qualification)
 Old master (*see* File maintenance)
 OPEN, 17, 54, 239
 Operator precedence rule (*see* Hierarchy)
 Optional reserved word, 42
 OR, 99-100
 ORGANIZATION IS INDEXED, 238, 242
 OUTPUT (*see* OPEN)

P

as CREDIT command, 32, 34
 as compiler option, 27
 Padding (*see* MOVE)
 PAGE, 54
 Paragraph, 4, 298
 Paragraph name, 5, 127, 128
 Parentheses:
 in arithmetic statement, 52
 in compound condition, 100
 PERFORM, 58-59, 76
 paragraphs vs. sections, 102-3, 134
 THRU, 103
 UNTIL, 18, 59
 VARYING, 111-12, 117, 175, 186
 with three level table, 197-98
 with two level table, 204
 Period, 6, 85, 87
 PICTURE, 7, 17, 44, 87
 Plus sign:
 in arithmetic expression, 52
 in PICTURE, 146, 147
 POSITION, 104, 105
 Prefixing (of dataname), 86, 143
 Priming read (*see* Initial read)
 PRINT utility, 30, 238, 242
 Procedure, 298
 PROCEDURE DIVISION, 3, 48-59
 PROGRAM-ID, 35, 43
 Programmer supplied name, 5
 Programming style, 125-43
 Program stub, 219-24
 Pseudocode, 14-15, 151, 156, 217-18, 244
 Punctuation character, 298
 Punctuation in COBOL, 6

Q

as CREDIT command, 32, 34
 Qualification, 107-9, 117, 298

R

as CREDIT command, 32
 RANDOM, 238, 242
 Range check, 229
 READ, 88
 AT END, 53, 130, 298
 INTO, 105
 INVALID KEY, 239, 245, 298
 Reasonableness check, 229
 Receiving item (*see* MOVE)

Record, 12, 298
 RECORD CONTAINS, 46, 87
 RECORD KEY, 239, 242, 245
 REDEFINES, 112-13, 186
 Relational operator, 298
 REPLACING (*see* INSPECT)
 Reserved word, 5, 42, 70, 75, 87, 276-79
 REVERSE, 104, 105
 REWRITE, 239
 ROUNDED, 106-7, 117
 RSCOBOL, 26, 27
 RUNCOBOL, 30

S

as CREDIT command, 32
 in PICTURE, 147-48
 SEARCH, 186
 SEARCH ALL, 186
 Section, 102, 134, 298
 with DECLARATIVES, 240
 Sector, 236
 Sector index, 237-38
 SECURITY, 43
 SELECT, 16, 19, 31, 44, 70, 238, 242
 Selection structure, 136-37
 Sequence check, 230
 Sequence numbers in COBOL, 34
 Sequence structure, 136-37
 Sequencing paragraph names, 128
 Sequential file maintenance (*see* File maintenance)
 SET, 188-89
 Signed number, 147-48, 151, 152
 SIZE ERROR, 106-7, 117, 130
 Slash:
 in column, 7, 130
 as editing character, 146
 SOURCE-COMPUTER, 7, 44
 Source program, 24
 SPACE, 47, 55
 Standards, 126-36
 START, 75
 STOP RUN, 59
 Structured English, 14
 Structured programming, 136-43
 Stub (*see* Program stub)
 Subprogram, 169-76
 Subroutine (*see* Subprogram)
 Subscript, 134-35 (*see* OCCURS and Table entries)
 SUBTRACT, 49-50, 130
 Switch, 134-35
 Symbol, 6
 Syntax, 42, 282-96

T

as CREDIT command, 32, 34
 as compiler option, 27
 TAB, 34
 Tables, 109-14, 184-204
 initialization, 112-13, 116, 187-88
 lookup, 113-14, 186-89
 one dimension, 110
 three dimension, 202-4

Tables (*cont.*)
 two dimension, 196-202
 variable length, 184-85, 187
 Testing (*see* Top down testing)
 Three-level table (*see* Table)
 THROUGH (*see* THRU)
 THRU (*see* PERFORM)
 TIME, 104-5
 Top down testing, 219-24
 Track, 230
 Track index, 236-38
 Transaction code, 216
 TRSDOS, 24, 26
 TRS-80 reference manuals, 24
 Truncation (*see* MOVE)
 Two-level table (*see* Table)

U

Unmatched transaction, 217
 UNTIL (*see* PERFORM)
 UP BY (*see* SET)
 Update program (*see* File maintenance)
 USE, 240
 USING, 170-71, 175

V

as assumed decimal point, 56-58
 VALUE clause, 47, 87, 106, 112, 186
 VALUES ARE, 100
 Variable-length table (*see* Table)
 VARYING (*see* PERFORM)

W

as CREDIT command, 32, 34
 WORKING-STORAGE SECTION, 46-47
 WRITE, 53-54, 88
 AFTER ADVANCING, 130
 FROM, 106
 INVALID KEY, 239

X

as CREDIT command, 32, 34
 as compiler option, 27
 in PICTURE clause, 7, 44

Y

Yourdon, E., 128

Z

as editing character, 64, 146, 147
 Zero:
 as editing character, 146
 ZERO, 47
 ZEROES, 47
 ZEROS, 47, 55

TRS-80 COBOL

Robert T. Grauer

TRS-80 COBOL is a substantial *COBOL* text covering all elements of the language, as implemented on the TRS-80 machines. **Robert T. Grauer** adheres to sound programming practices, and uses structured programming exclusively. Pseudocode and hierarchy charts are emphasized, while the traditional flowchart is de-emphasized.

This is a "learn by doing" approach that stresses early access to the machine as well as constant exposure to complete *COBOL* programs. Every *COBOL* chapter contains at least one program to tie together the major points in that chapter.

A listing of chapter contents shows the wide range of material:

- Chapter 1 is a rapid introduction to *COBOL*.
- Chapter 2 develops concepts of file processing.
- Chapter 3 deals exclusively with the TRS-80.
- Chapter 4 returns to *COBOL*.
- Chapter 5 debugs both compilation and execution errors.
- Chapter 6 introduces some advanced elements of the language.
- Chapter 7 discusses programming style.
- Chapter 8 introduces the concept of control breaks.
- Chapter 9 covers subprograms and the *COPY* statement.
- Chapter 10 is a comprehensive treatment of table processing.
- Chapter 11 focuses on sequential file maintenance.
- Chapter 12 parallels Chapter 11, except for nonsequential maintenance.

Review exercises are included at the end of each chapter.